EXHIBIT G

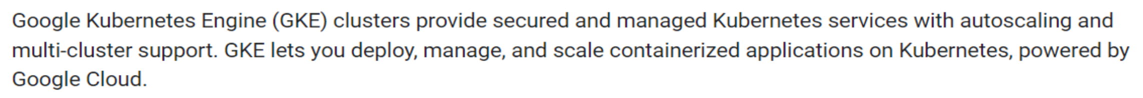
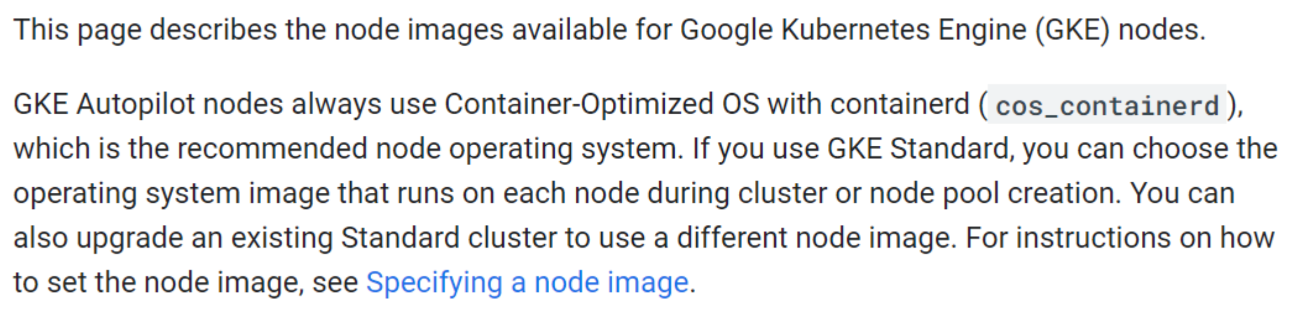
U.S. Patent No. 7,519,814 ("'814 Patent")

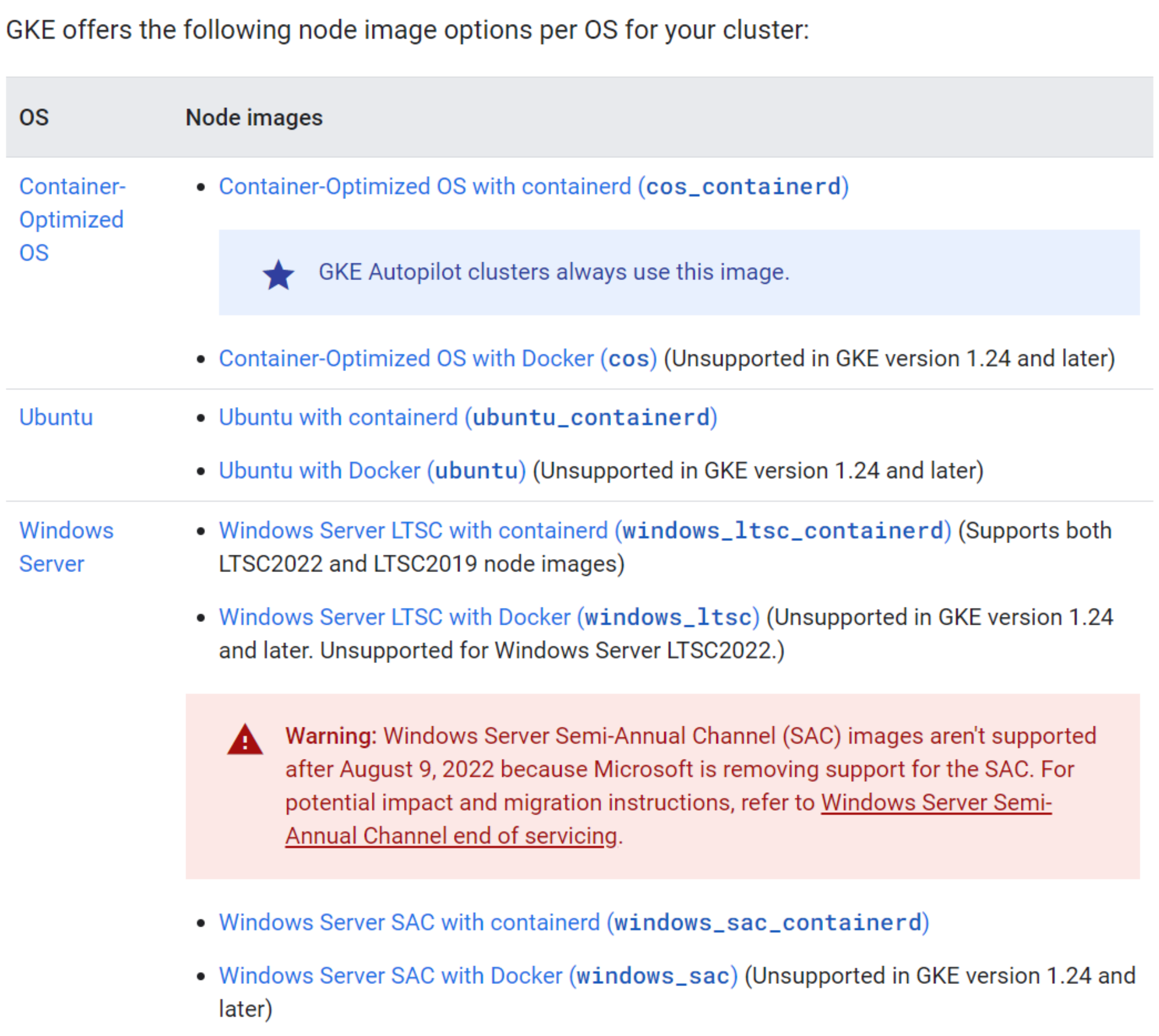
Accused Instrumentalities: Google's products and services using user mode critical system elements as shared libraries, including without limitation Google Kubernetes Engine, Cloud Run, Migrate to Containers, and all versions and variations thereof since the issuance of the asserted patent. To the extent Google contends "Google Cloud Observability" is a separate instrumentality, the combination of GKE and Google Cloud Observability and the combination of Cloud Run and Google Cloud Observability are Accused Instrumentalities with respect to claim 13.

Each Accused Instrumentality infringes the claims in substantially the same way, and the evidence shown in this chart is similarly applicable to each Accused Instrumentality. Each claim limitation is literally infringed by each Accused Instrumentality. However, to the extent any claim limitation is not met literally, it is nonetheless met under the doctrine of equivalents because the differences between the claim limitation and each Accused Instrumentality would be insubstantial, and each Accused Instrumentality performs substantially the same function, in substantially the same way, to achieve the same result as the claimed invention. Notably, Defendant has not yet articulated which, if any, particular claim limitations it believes are not met by the Accused Instrumentalities.

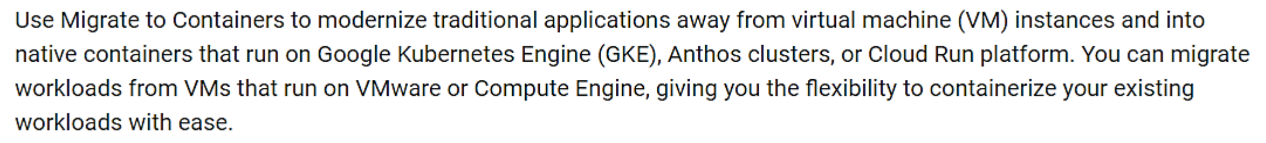
Claim 1

Claim 1	Accused Instrumentalities
[1pre] 1. In a system having a plurality of servers with operating systems that differ, operating in disparate computing environments, wherein each server includes a processor and an operating system including a kernel a set of associated local system files compatible with the processor, a method of providing at least some of the servers in the system with secure, executable, applications related to a service, wherein the applications are executed in a	To the extent the preamble is limiting, Google and/or its customer practices, through the Accused Instrumentalities, in a system having a plurality of servers with operating systems that differ, operating in disparate computing environments, wherein each server includes a processor and an operating system including a kernel a set of associated local system files compatible with the processor, a method of providing at least some of the servers in the system with secure, executable, applications related to a service, wherein the applications are executed in a secure environment, wherein the applications each include an object executable by at least some of the different operating systems for performing a task related to the service, as claimed. For example, Google Kubernetes Engine and Cloud Run,~~ as well as containers produced by Migrate to Containers,~~ each runs on individual servers, each of which uses an independent operating system. This is also true in the infringing configuration where Migrate to Containers is used to produce a container that is run on Google Kubernetes Engine or Cloud Run. Google provides and/or requires that each server includes a processor with one or more cores available to the OS kernel. Google further provides and/or requires that each server has a supported operating system (*e.g.*, Container-Optimized OS, Ubuntu), which includes a kernel and associated local system files, including for example libraries such as libc/glibc, configuration files, etc. On information and belief, ~~there exist at~~

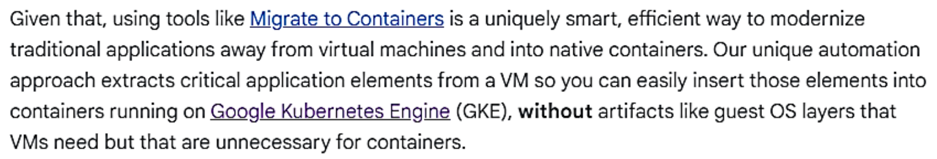
Claim 1	Accused Instrumentalities
secure environment, wherein the applications each include an object executable by at least some of the different operating systems for performing a task related to the service, the method comprising:	~~least two~~ GKE/Cloud Run servers ~~that~~ have different operating systems, for example Container-Optimized OS and Ubuntu. The servers operate in disparate computing environments, including because each server is a stand-alone computer and/or each server is unrelated to the other servers due to having independent hardware and, in some instances, independent software. *See* claim limitations below. *See also, e.g.*: Google Kubernetes Engine (GKE) clusters provide secured and managed Kubernetes services with autoscaling and multi-cluster support. GKE lets you deploy, manage, and scale containerized applications on Kubernetes, powered by Google Cloud. https://cloud.google.com/migrate/containers/docs/getting-started This page describes the node images available for Google Kubernetes Engine (GKE) nodes. GKE Autopilot nodes always use Container-Optimized OS with containerd (`cos_containerd`), which is the recommended node operating system. If you use GKE Standard, you can choose the operating system image that runs on each node during cluster or node pool creation. You can also upgrade an existing Standard cluster to use a different node image. For instructions on how to set the node image, see Specifying a node image. https://cloud.google.com/kubernetes-engine/docs/concepts/node-images

Claim 1	Accused Instrumentalities
	 GKE offers the following node image options per OS for your cluster: **OS** / **Node images** Container-Optimized OS: • Container-Optimized OS with containerd (`cos_containerd`) ★ GKE Autopilot clusters always use this image. • Container-Optimized OS with Docker (`cos`) (Unsupported in GKE version 1.24 and later) Ubuntu: • Ubuntu with containerd (`ubuntu_containerd`) • Ubuntu with Docker (`ubuntu`) (Unsupported in GKE version 1.24 and later) Windows Server: • Windows Server LTSC with containerd (`windows_ltsc_containerd`) (Supports both LTSC2022 and LTSC2019 node images) • Windows Server LTSC with Docker (`windows_ltsc`) (Unsupported in GKE version 1.24 and later. Unsupported for Windows Server LTSC2022.) **Warning:** Windows Server Semi-Annual Channel (SAC) images aren't supported after August 9, 2022 because Microsoft is removing support for the SAC. For potential impact and migration instructions, refer to Windows Server Semi-Annual Channel end of servicing. • Windows Server SAC with containerd (`windows_sac_containerd`) • Windows Server SAC with Docker (`windows_sac`) (Unsupported in GKE version 1.24 and later) https://cloud.google.com/kubernetes-engine/docs/concepts/node-images

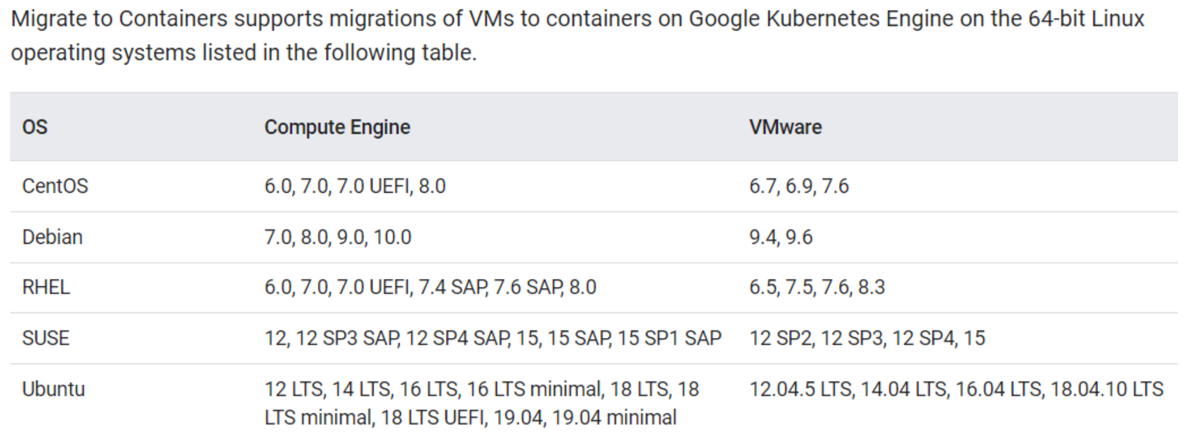
Claim 1	Accused Instrumentalities


Use Migrate to Containers to modernize traditional applications away from virtual machine (VM) instances and into native containers that run on Google Kubernetes Engine (GKE), Anthos clusters, or Cloud Run platform. You can migrate workloads from VMs that run on VMware or Compute Engine, giving you the flexibility to containerize your existing workloads with ease.

https://cloud.google.com/migrate/containers/docs/getting-started

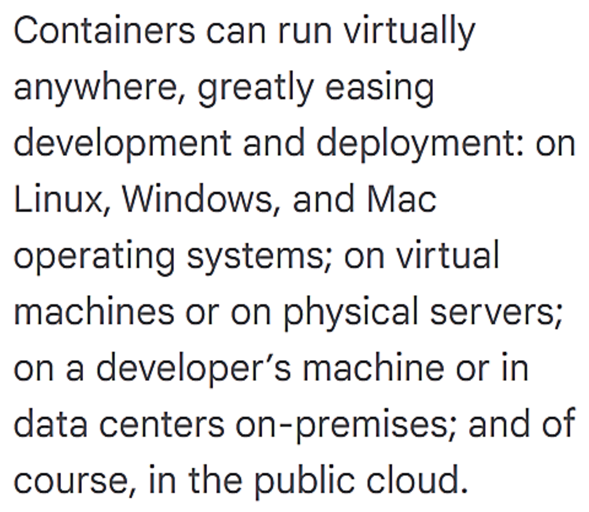
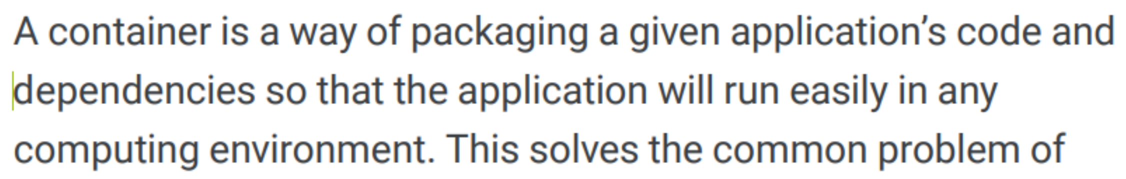

Given that, using tools like Migrate to Containers is a uniquely smart, efficient way to modernize traditional applications away from virtual machines and into native containers. Our unique automation approach extracts critical application elements from a VM so you can easily insert those elements into containers running on Google Kubernetes Engine (GKE), **without** artifacts like guest OS layers that VMs need but that are unnecessary for containers.

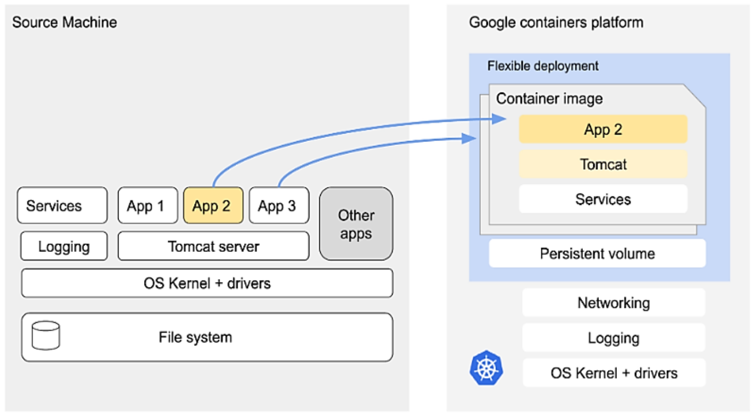
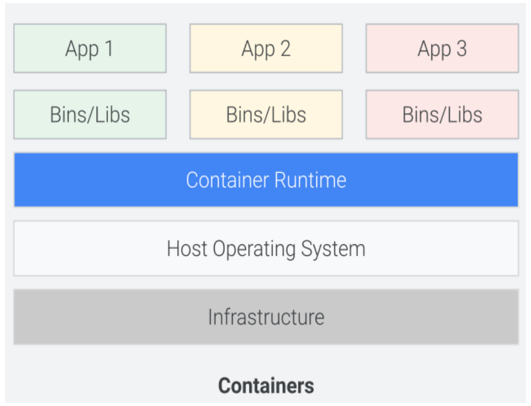
https://cloud.google.com/blog/products/containers-kubernetes/how-migrate-for-anthos-improves-vm-to-container-migration

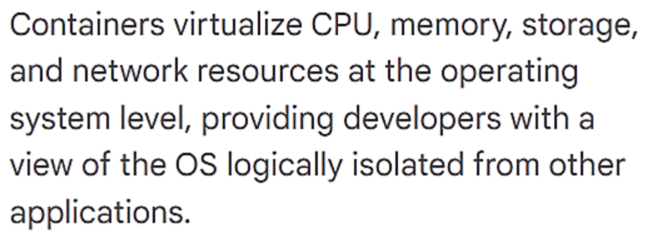
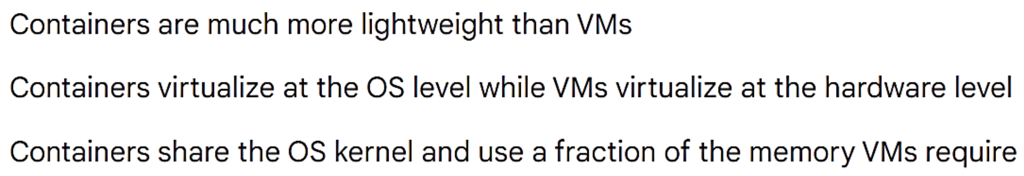
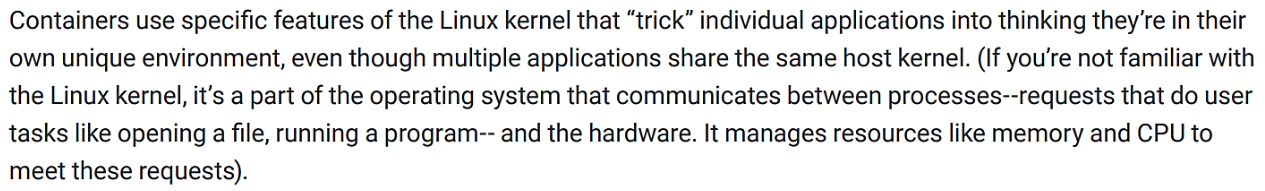
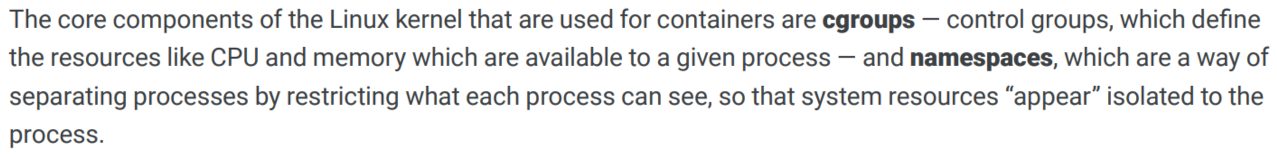

Migrate to Containers supports migrations of VMs to containers on Google Kubernetes Engine on the 64-bit Linux operating systems listed in the following table.

OS	Compute Engine	VMware
CentOS	6.0, 7.0, 7.0 UEFI, 8.0	6.7, 6.9, 7.6
Debian	7.0, 8.0, 9.0, 10.0	9.4, 9.6
RHEL	6.0, 7.0, 7.0 UEFI, 7.4 SAP, 7.6 SAP, 8.0	6.5, 7.5, 7.6, 8.3
SUSE	12, 12 SP3 SAP, 12 SP4 SAP, 15, 15 SAP, 15 SP1 SAP	12 SP2, 12 SP3, 12 SP4, 15
Ubuntu	12 LTS, 14 LTS, 16 LTS, 16 LTS minimal, 18 LTS, 18 LTS minimal, 18 LTS UEFI, 19.04, 19.04 minimal	12.04.5 LTS, 14.04 LTS, 16.04 LTS, 18.04.10 LTS

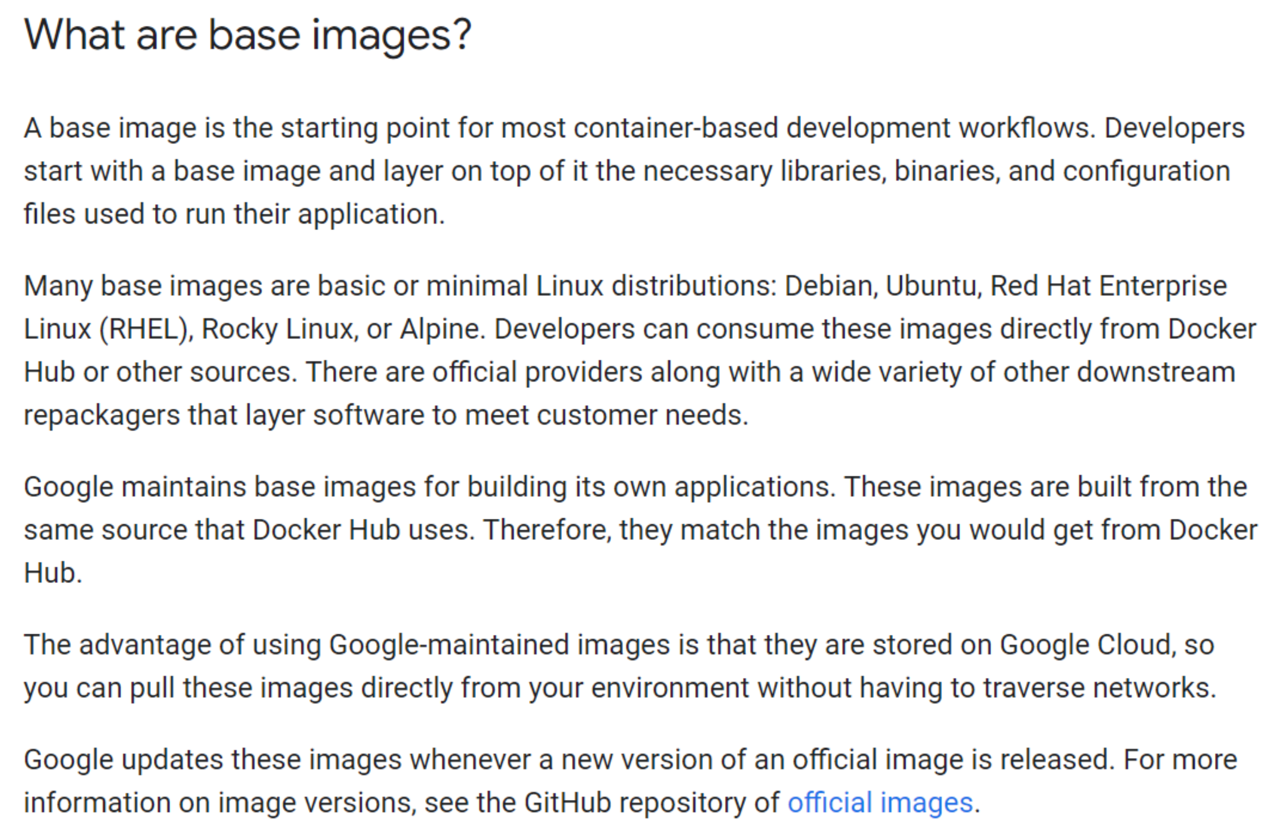
https://cloud.google.com/migrate/containers/docs/compatible-os-versions, Last accessed on June 05, 2023

Claim 1	Accused Instrumentalities
	 Containers can run virtually anywhere, greatly easing development and deployment: on Linux, Windows, and Mac operating systems; on virtual machines or on physical servers; on a developer's machine or in data centers on-premises; and of course, in the public cloud. https://cloud.google.com/learn/what-are-containers  A container is a way of packaging a given application's code and dependencies so that the application will run easily in any computing environment. This solves the common problem of https://services.google.com/fh/files/misc/why_container_security_matters.pdf

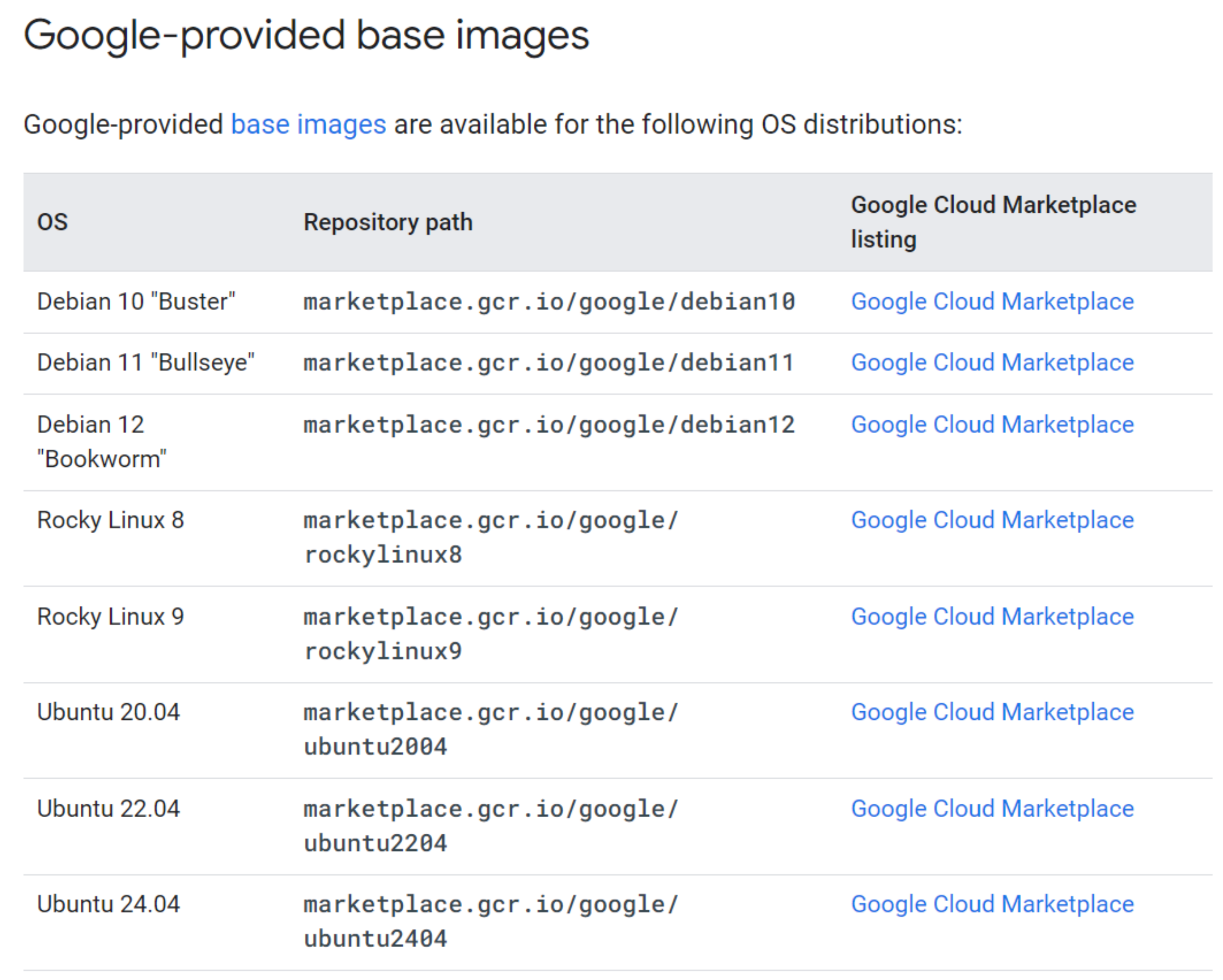
Claim 1	Accused Instrumentalities
	 Source Machine Services App 1 App 2 App 3 Other apps Logging Tomcat server OS Kernel + drivers File system Google containers platform Flexible deployment Container image App 2 Tomcat Services Persistent volume Networking Logging OS Kernel + drivers https://cloud.google.com/blog/products/application-modernization/shift-your-apps-to-container-based-workloads-on-the-command-line  App 1 App 2 App 3 Bins/Libs Bins/Libs Bins/Libs Container Runtime Host Operating System Infrastructure Containers https://services.google.com/fh/files/misc/why_container_security_matters.pdf

Claim 1	Accused Instrumentalities
	 Containers virtualize CPU, memory, storage, and network resources at the operating system level, providing developers with a view of the OS logically isolated from other applications. https://cloud.google.com/learn/what-are-containers  Containers are much more lightweight than VMs Containers virtualize at the OS level while VMs virtualize at the hardware level Containers share the OS kernel and use a fraction of the memory VMs require https://cloud.google.com/learn/what-are-containers  Containers use specific features of the Linux kernel that “trick” individual applications into thinking they’re in their own unique environment, even though multiple applications share the same host kernel. (If you’re not familiar with the Linux kernel, it’s a part of the operating system that communicates between processes--requests that do user tasks like opening a file, running a program-- and the hardware. It manages resources like memory and CPU to meet these requests). https://services.google.com/fh/files/misc/why_container_security_matters.pdf  The core components of the Linux kernel that are used for containers are **cgroups** — control groups, which define the resources like CPU and memory which are available to a given process — and **namespaces**, which are a way of separating processes by restricting what each process can see, so that system resources “appear” isolated to the process. https://services.google.com/fh/files/misc/why_container_security_matters.pdf

Claim 1	Accused Instrumentalities
[1a] storing in memory accessible to at least some of the servers a plurality of secure containers of application software, each container comprising one or more of the executable applications and a set of associated system files required to execute the one or more applications, for use with a local kernel residing permanently on one of the servers;	The method practiced by Google and/or its customer through the Accused Instrumentalities includes a step of storing in memory accessible to at least some of the servers a plurality of secure containers of application software, each container comprising one or more of the executable applications and a set of associated system files required to execute the one or more applications, for use with a local kernel residing permanently on one of the servers. For example, GKE and~~,~~ Cloud Run~~, and Migrate to Containers~~ each stores application containers, sometimes called Docker containers, container images, Kubernetes containers, or Kubernetes pods, in persistent storage available to each node running the application. This is also true in the infringing configuration where Migrate to Containers is used to produce a container that is run on Google Kubernetes Engine or Cloud Run. The terms "node" and "host" are sometimes used to refer to the claimed server. The container might be in a format defined by the Open Container Initiative. This storage may be physically attached to the server or connected through any supported interconnect, including over a network. In addition to ~~Each container includes~~ the application software, each container includes associated system files, including ~~as well as~~ a Linux user space required to execute the application, for example libc/glibc and other shared libraries, configuration files, etc. necessary for the application. For example, the container includes a base OS image, provided by Google or by a third party, such as a Debian, Rocky Linux, or Ubuntu base image. The container is compatible with the host kernel, for example because the container libraries are linked against the Linux kernel, and the supported host operating systems also use the Linux kernel, which has a stable binary interface. For another example, GKE and Cloud Run each stores files, pertaining to the applications, in ephemeral or persistent volumes or in the filesystem represented in the container image, required to execute the applications within those containers. Because these volumes are stored and accessible within the GKE/Cloud Run environment, it is inferred that they are stored in the memory of the server as claimed. The containers are secure containers as claimed. For example, the data within an individual container is insulated from the effects of other containers except to the extent the container is specifically configured to allow other containers to modify its data, for example using a shared volume. *See, e.g.*:

Claim 1	Accused Instrumentalities
	 What are base images? A base image is the starting point for most container-based development workflows. Developers start with a base image and layer on top of it the necessary libraries, binaries, and configuration files used to run their application. Many base images are basic or minimal Linux distributions: Debian, Ubuntu, Red Hat Enterprise Linux (RHEL), Rocky Linux, or Alpine. Developers can consume these images directly from Docker Hub or other sources. There are official providers along with a wide variety of other downstream repackagers that layer software to meet customer needs. Google maintains base images for building its own applications. These images are built from the same source that Docker Hub uses. Therefore, they match the images you would get from Docker Hub. The advantage of using Google-maintained images is that they are stored on Google Cloud, so you can pull these images directly from your environment without having to traverse networks. Google updates these images whenever a new version of an official image is released. For more information on image versions, see the GitHub repository of official images. https://cloud.google.com/software-supply-chain-security/docs/base-images

Claim 1	Accused Instrumentalities

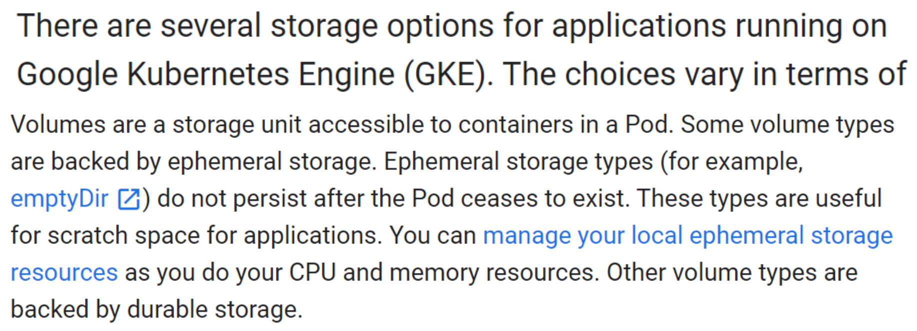
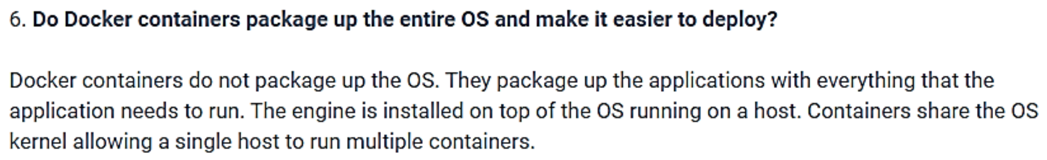
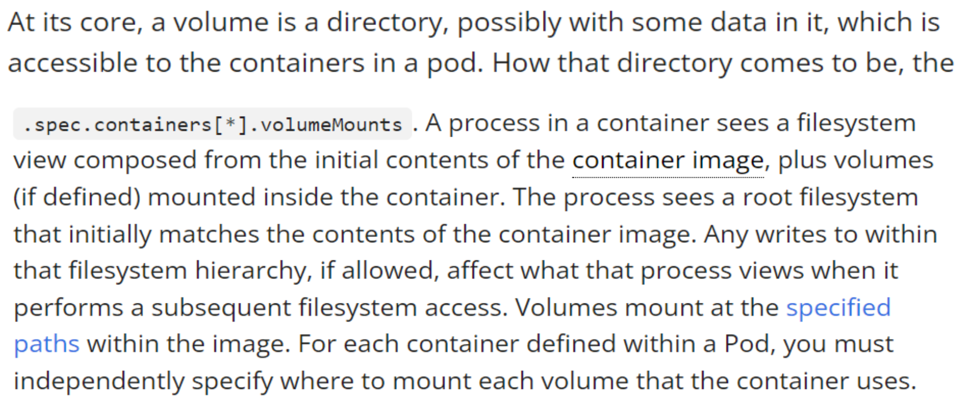


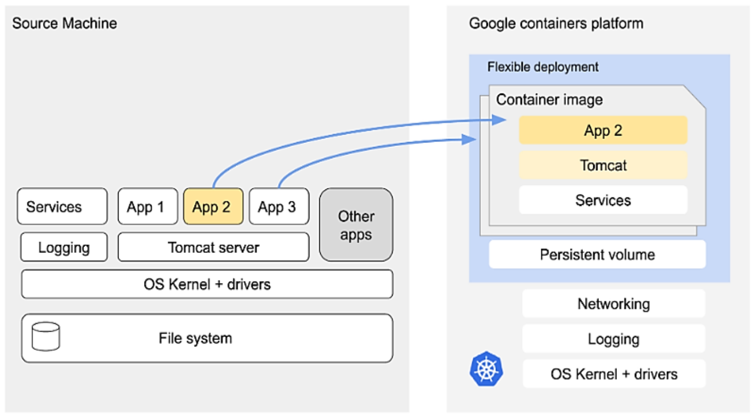
Google-provided base images

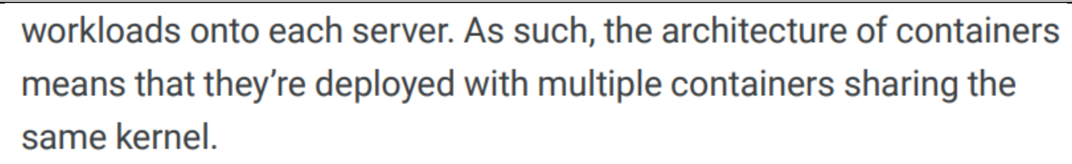
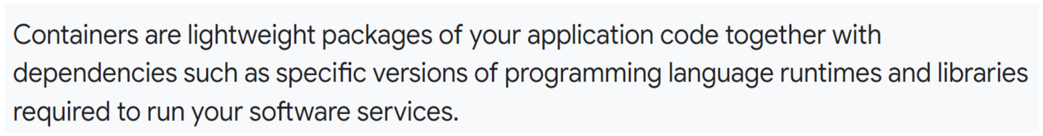
Google-provided base images are available for the following OS distributions:

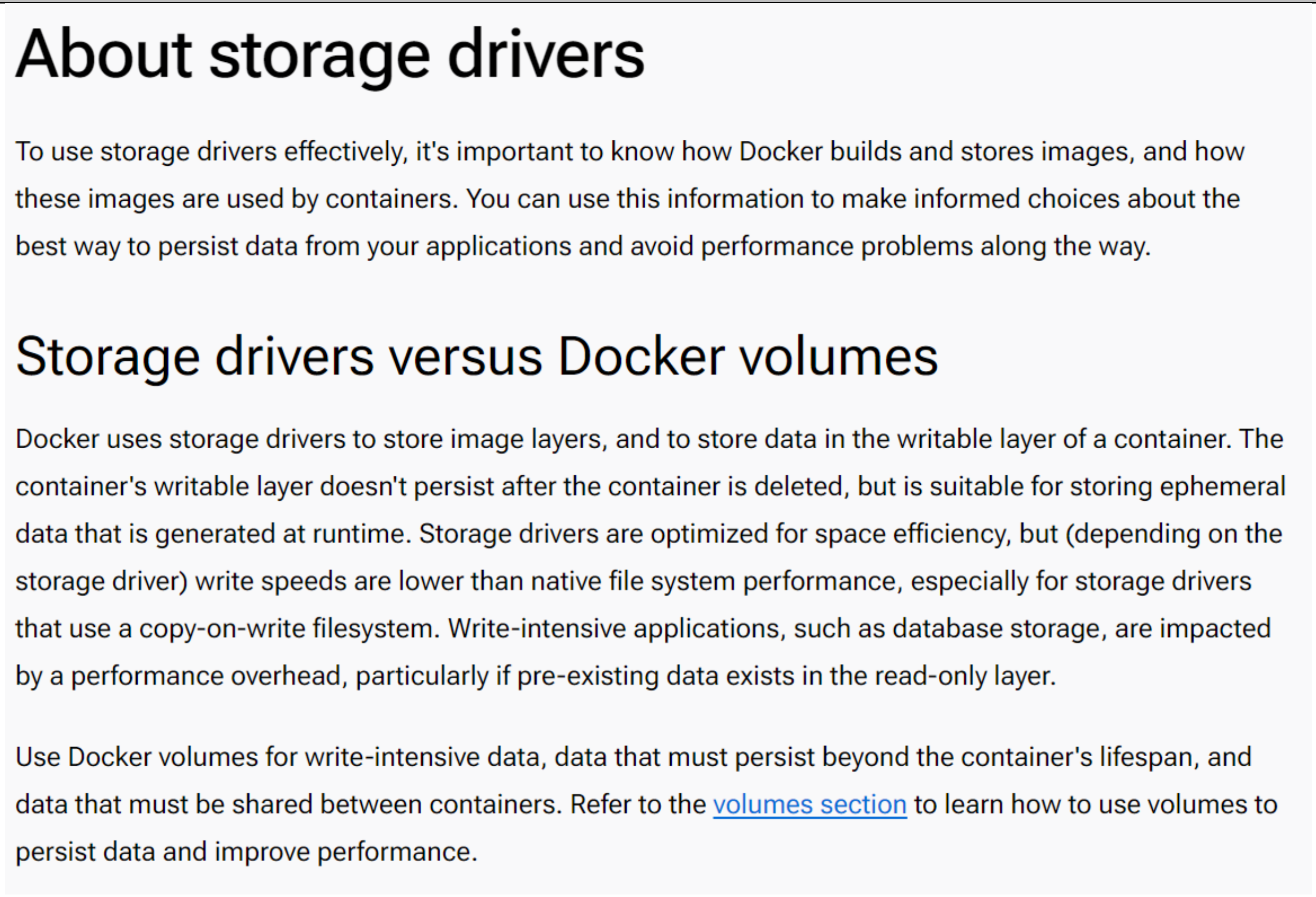
OS	Repository path	Google Cloud Marketplace listing
Debian 10 "Buster"	`marketplace.gcr.io/google/debian10`	Google Cloud Marketplace
Debian 11 "Bullseye"	`marketplace.gcr.io/google/debian11`	Google Cloud Marketplace
Debian 12 "Bookworm"	`marketplace.gcr.io/google/debian12`	Google Cloud Marketplace
Rocky Linux 8	`marketplace.gcr.io/google/rockylinux8`	Google Cloud Marketplace
Rocky Linux 9	`marketplace.gcr.io/google/rockylinux9`	Google Cloud Marketplace
Ubuntu 20.04	`marketplace.gcr.io/google/ubuntu2004`	Google Cloud Marketplace
Ubuntu 22.04	`marketplace.gcr.io/google/ubuntu2204`	Google Cloud Marketplace
Ubuntu 24.04	`marketplace.gcr.io/google/ubuntu2404`	Google Cloud Marketplace

https://cloud.google.com/software-supply-chain-security/docs/base-images

Claim 1	Accused Instrumentalities
	 There are several storage options for applications running on Google Kubernetes Engine (GKE). The choices vary in terms of Volumes are a storage unit accessible to containers in a Pod. Some volume types are backed by ephemeral storage. Ephemeral storage types (for example, emptyDir) do not persist after the Pod ceases to exist. These types are useful for scratch space for applications. You can manage your local ephemeral storage resources as you do your CPU and memory resources. Other volume types are backed by durable storage. https://cloud.google.com/kubernetes-engine/docs/concepts/storage-overview  6. **Do Docker containers package up the entire OS and make it easier to deploy?** Docker containers do not package up the OS. They package up the applications with everything that the application needs to run. The engine is installed on top of the OS running on a host. Containers share the OS kernel allowing a single host to run multiple containers. https://www.docker.com/blog/the-10-most-common-questions-it-admins-ask-about-docker/  At its core, a volume is a directory, possibly with some data in it, which is accessible to the containers in a pod. How that directory comes to be, the `.spec.containers[*].volumeMounts` . A process in a container sees a filesystem view composed from the initial contents of the container image, plus volumes (if defined) mounted inside the container. The process sees a root filesystem that initially matches the contents of the container image. Any writes to within that filesystem hierarchy, if allowed, affect what that process views when it performs a subsequent filesystem access. Volumes mount at the specified paths within the image. For each container defined within a Pod, you must independently specify where to mount each volume that the container uses. https://kubernetes.io/docs/concepts/storage/volumes/

Claim 1	Accused Instrumentalities
	 Source Machine Services App 1 App 2 App 3 Other apps Logging Tomcat server OS Kernel + drivers File system Google containers platform Flexible deployment Container image App 2 Tomcat Services Persistent volume Networking Logging OS Kernel + drivers https://cloud.google.com/blog/products/application-modernization/shift-your-apps-to-container-based-workloads-on-the-command-line A container is a way of packaging a given application's code and dependencies so that the application will run easily in any computing environment. This solves the common problem of The container image specifies the container's file system. For example, if you're running a Node.js application, the container image would contain your app, Node.js, and other dependencies like Linux system libraries (except the kernel). A container image usually extends a base operating system image, or **base image**. This base image is the basis of your container, so you'll want to ensure that it's properly patched and free from known vulnerabilities.

Claim 1	Accused Instrumentalities
	 workloads onto each server. As such, the architecture of containers means that they're deployed with multiple containers sharing the same kernel. https://services.google.com/fh/files/misc/why_container_security_matters.pdf  Containers are lightweight packages of your application code together with dependencies such as specific versions of programming language runtimes and libraries required to run your software services. https://cloud.google.com/learn/what-are-containers

<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

About storage drivers

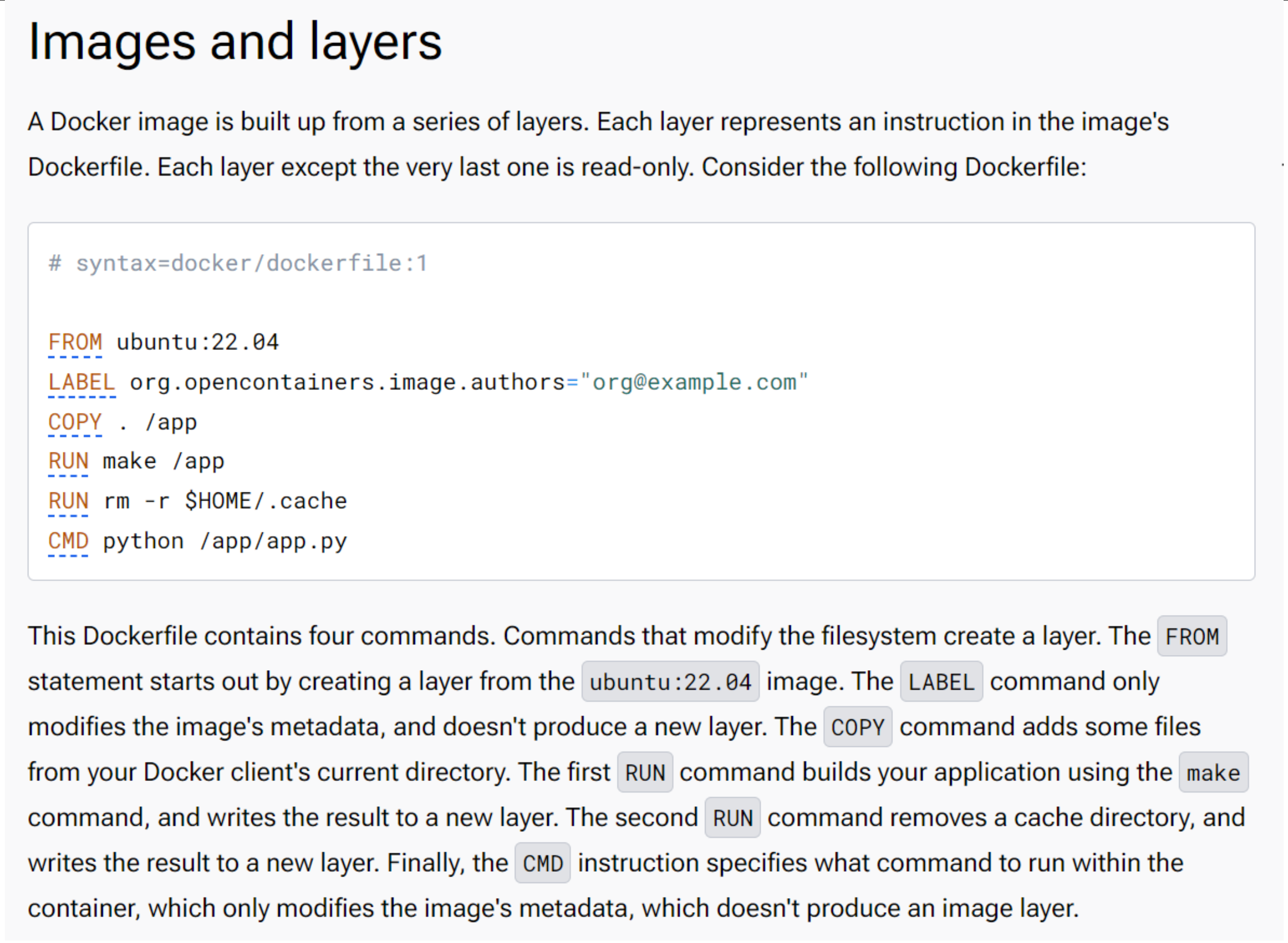
To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way.

Storage drivers versus Docker volumes

Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer.

Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance.

https://docs.docker.com/storage/storagedriver/</td></tr>
</table>

Claim 1	Accused Instrumentalities
	 https://docs.docker.com/storage/storagedriver/

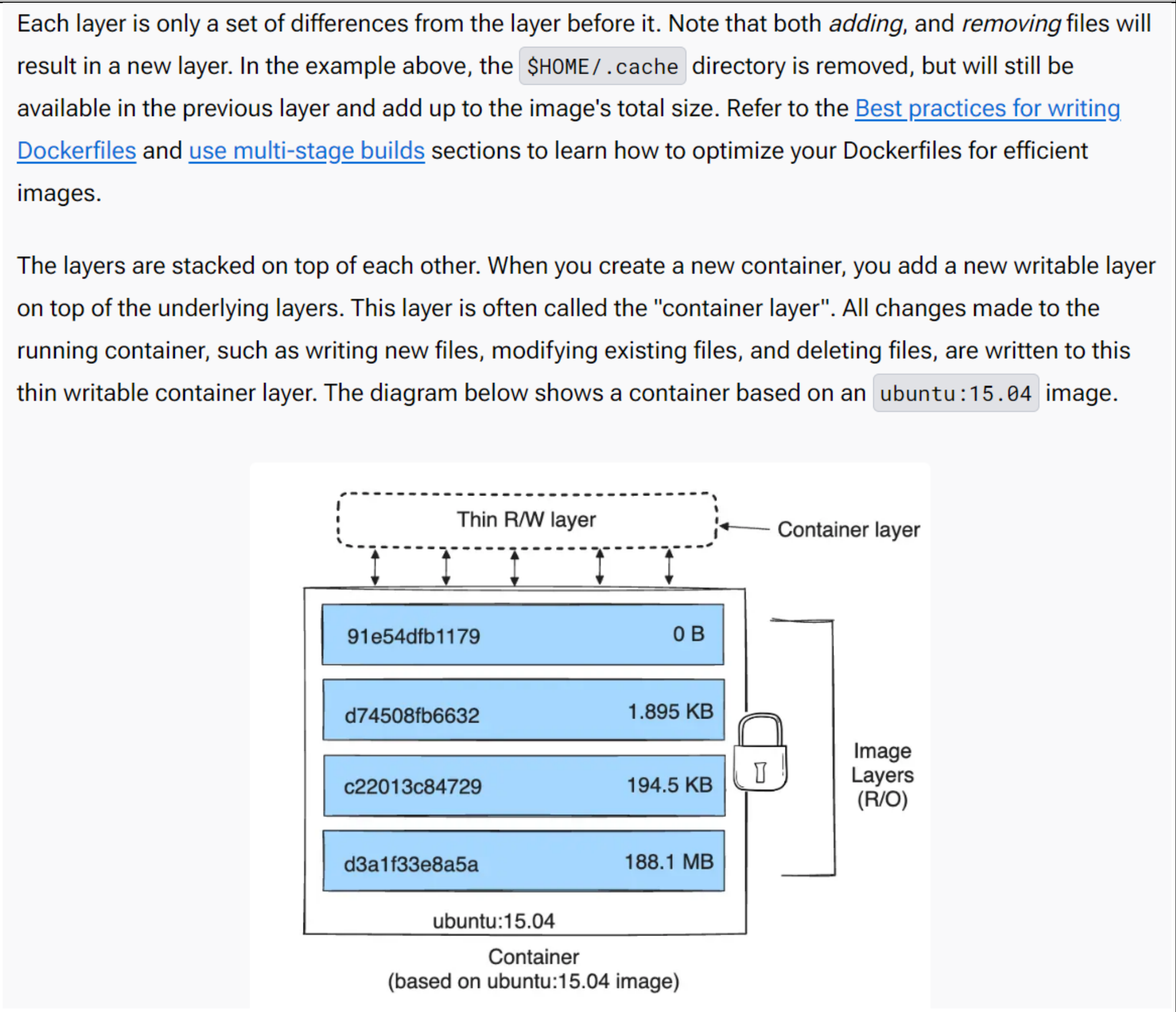
Images and layers

A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:

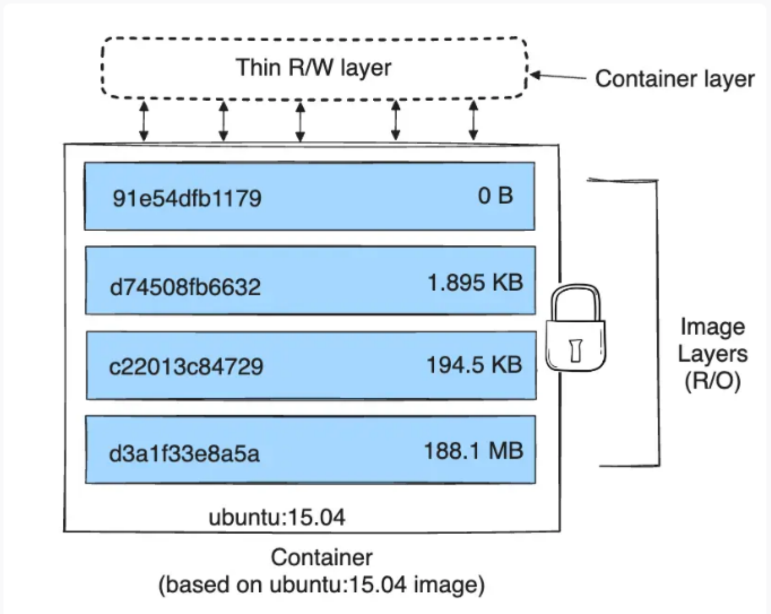
```
# syntax=docker/dockerfile:1

FROM ubuntu:22.04
LABEL org.opencontainers.image.authors="org@example.com"
COPY . /app
RUN make /app
RUN rm -r $HOME/.cache
CMD python /app/app.py
```

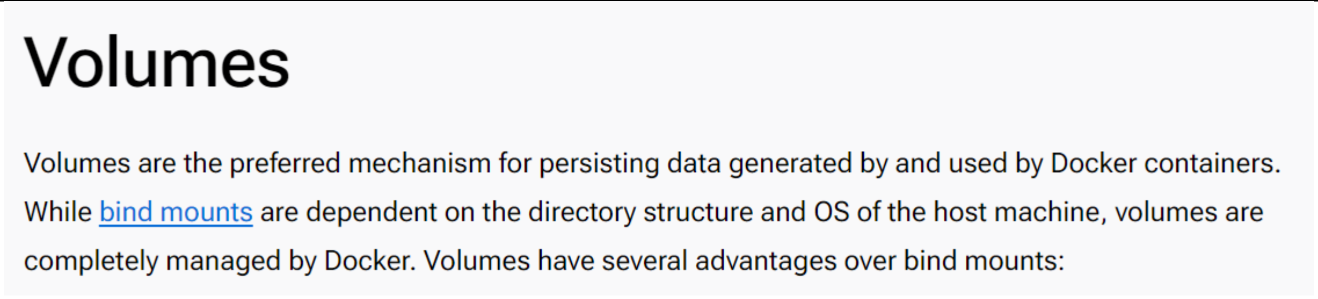
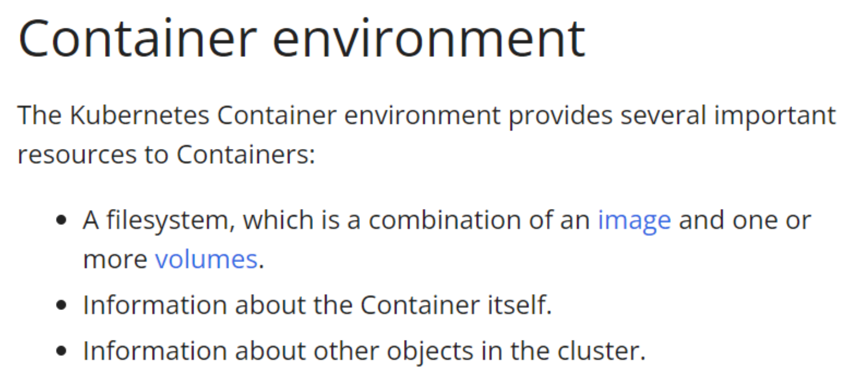
This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.

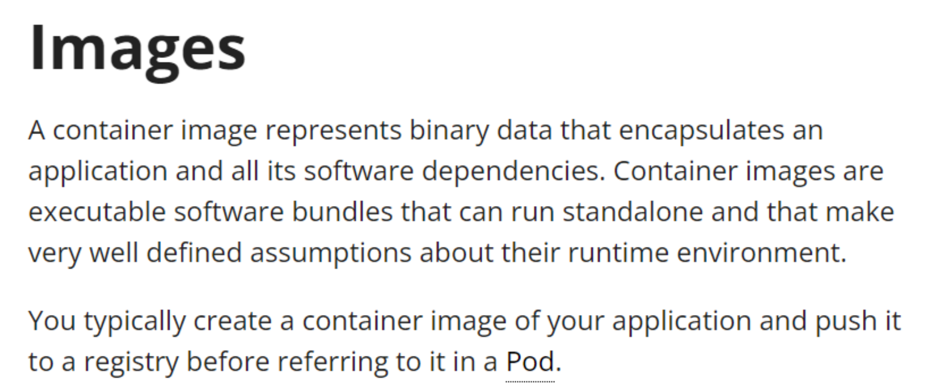
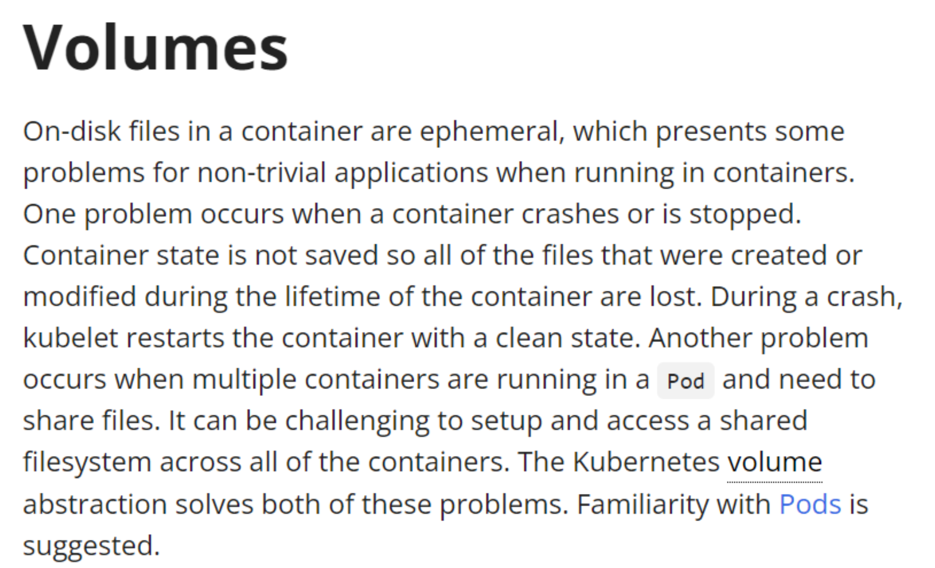
<table>
<tr><th>Claim 1</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>

Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images.

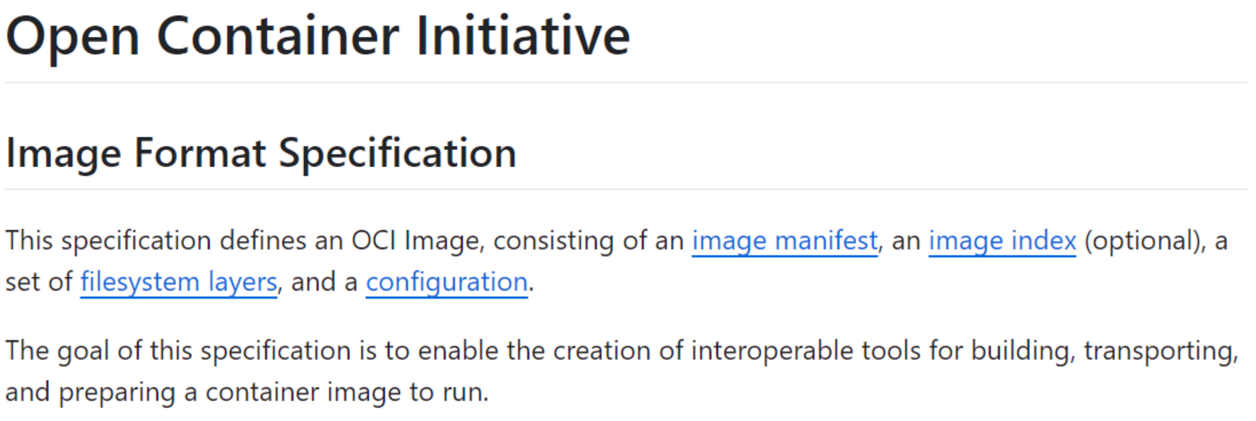
The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.

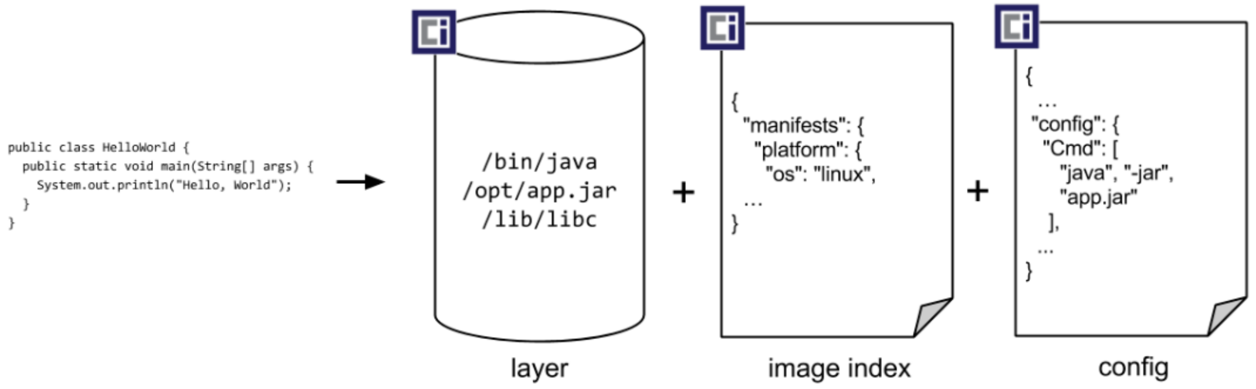


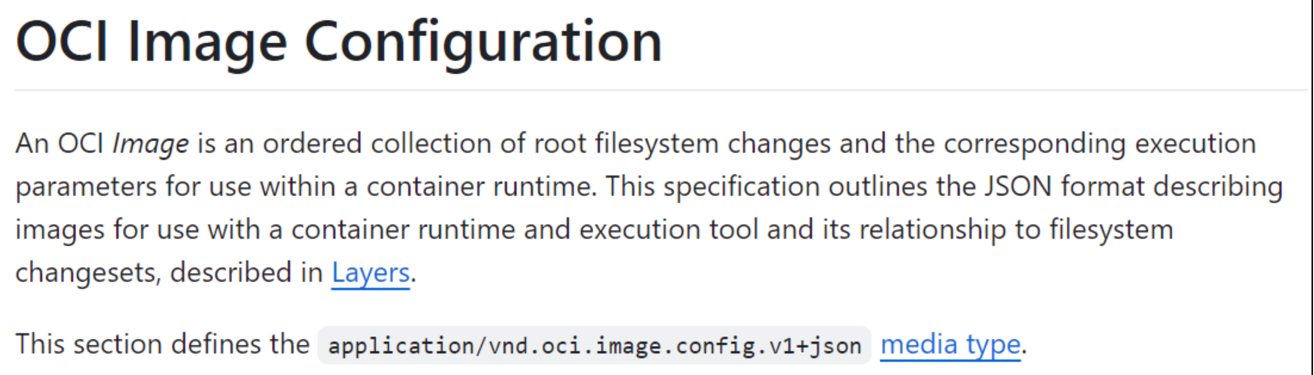
https://docs.docker.com/storage/storagedriver/
</td></tr>
</table>

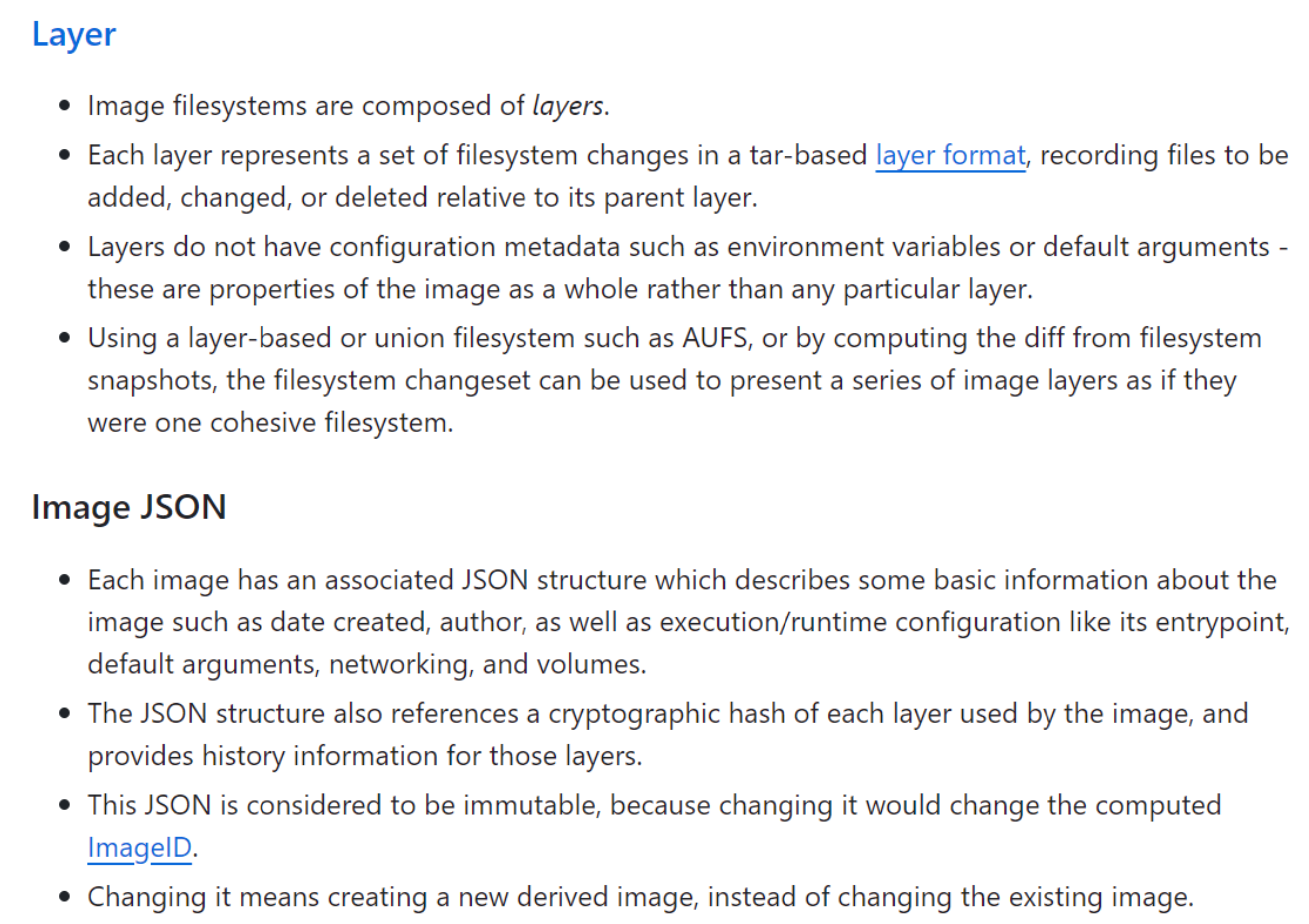
Claim 1	Accused Instrumentalities
	 **Volumes** Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure and OS of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts: https://kubernetes.io/docs/concepts/storage/volumes/  **Container environment** The Kubernetes Container environment provides several important resources to Containers: • A filesystem, which is a combination of an image and one or more volumes. • Information about the Container itself. • Information about other objects in the cluster. https://kubernetes.io/docs/concepts/containers/container-environment/

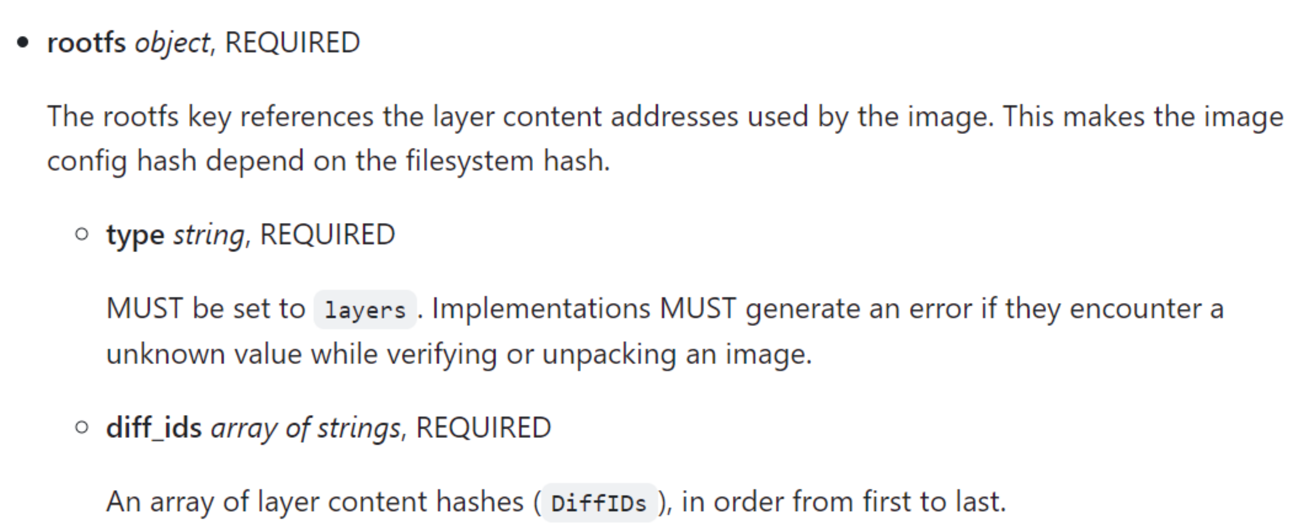
Claim 1	Accused Instrumentalities
	 **Images** A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment. You typically create a container image of your application and push it to a registry before referring to it in a Pod. https://kubernetes.io/docs/concepts/containers/images/  **Volumes** On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested. https://kubernetes.io/docs/concepts/storage/volumes/

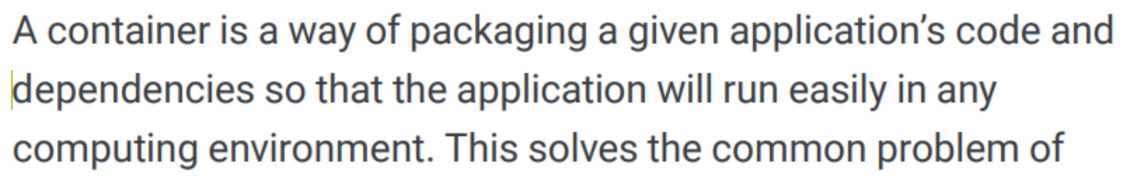
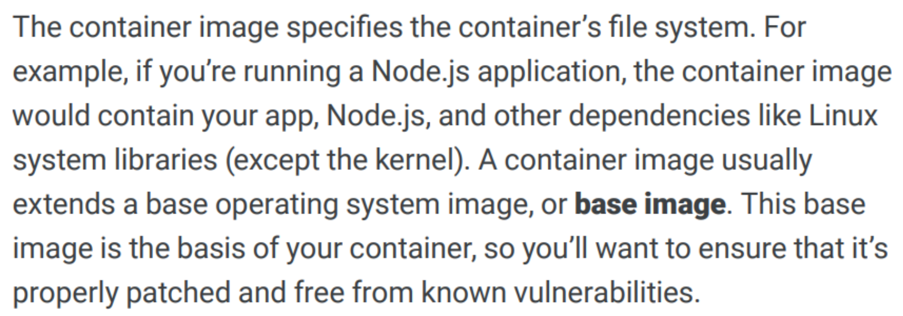
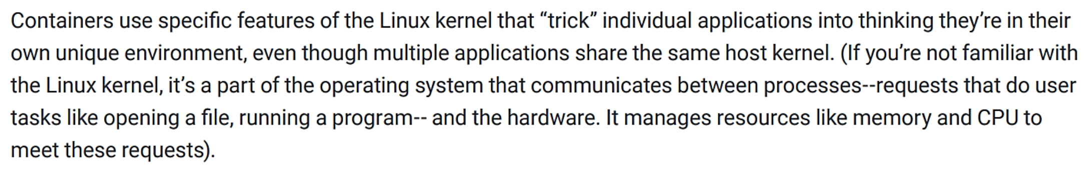
Claim 1	Accused Instrumentalities
	 Open Container Initiative Image Format Specification This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

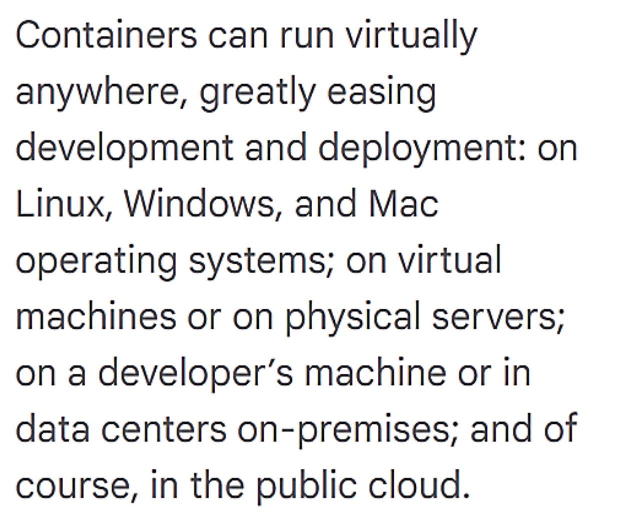
Claim 1	Accused Instrumentalities
	Overview At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.  public class HelloWorld { public static void main(String[] args) { System.out.println("Hello, World"); } } /bin/java /opt/app.jar /lib/libc layer { "manifests": { "platform": { "os": "linux", ... } image index { ... "config": { "Cmd": ["java", "-jar", "app.jar"], ... } config https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

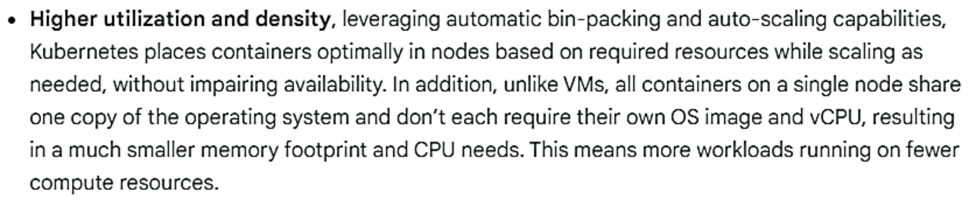
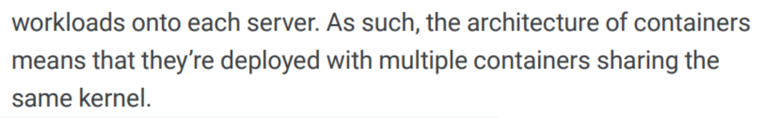
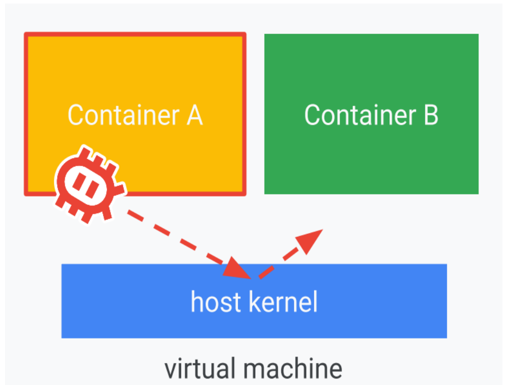
Claim 1	Accused Instrumentalities
	 OCI Image Configuration An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

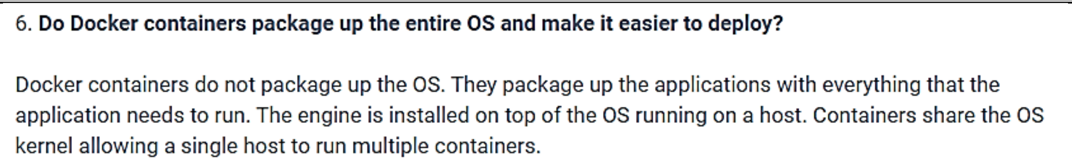
Claim 1	Accused Instrumentalities
	 **Layer** • Image filesystems are composed of *layers*. • Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer. • Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer. • Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem. **Image JSON** • Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes. • The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers. • This JSON is considered to be immutable, because changing it would change the computed ImageID. • Changing it means creating a new derived image, instead of changing the existing image. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

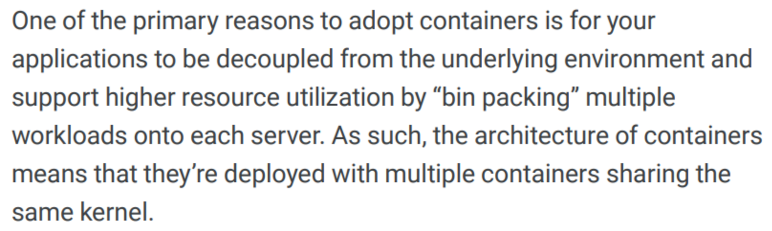
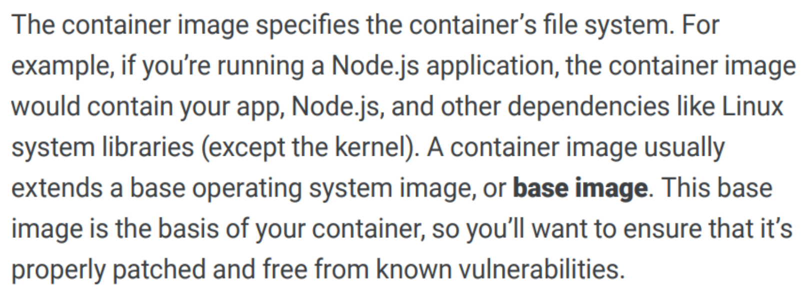
Claim 1	Accused Instrumentalities
	 • **rootfs** *object*, REQUIRED The rootfs key references the layer content addresses used by the image. This makes the image config hash depend on the filesystem hash. ○ **type** *string*, REQUIRED MUST be set to `layers`. Implementations MUST generate an error if they encounter a unknown value while verifying or unpacking an image. ○ **diff_ids** *array of strings*, REQUIRED An array of layer content hashes (`DiffIDs`), in order from first to last. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

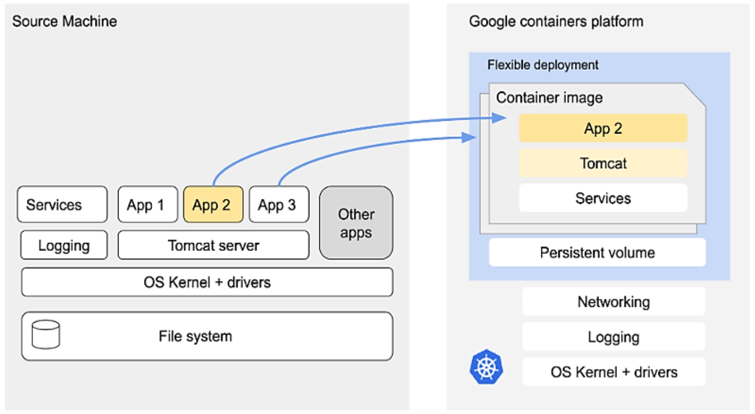
Claim 1	Accused Instrumentalities
[1b] wherein the set of associated system files are compatible with a local kernel of at least some of the plurality of different operating systems,	In the method practiced by Google and/or its customer through the Accused Instrumentalities, the set of associated system files are compatible with a local kernel of at least some of the plurality of different operating systems. The associated system files in the container are compatible with the host kernel, for example because they are linked against the Linux kernel and the supported host operating systems also use the Linux kernel, which has a stable binary interface. *See* discussion and evidence in element [1a] above. *See also, e.g.*:  A container is a way of packaging a given application's code and dependencies so that the application will run easily in any computing environment. This solves the common problem of  The container image specifies the container's file system. For example, if you're running a Node.js application, the container image would contain your app, Node.js, and other dependencies like Linux system libraries (except the kernel). A container image usually extends a base operating system image, or **base image**. This base image is the basis of your container, so you'll want to ensure that it's properly patched and free from known vulnerabilities.  Containers use specific features of the Linux kernel that "trick" individual applications into thinking they're in their own unique environment, even though multiple applications share the same host kernel. (If you're not familiar with the Linux kernel, it's a part of the operating system that communicates between processes--requests that do user tasks like opening a file, running a program-- and the hardware. It manages resources like memory and CPU to meet these requests). https://services.google.com/fh/files/misc/why_container_security_matters.pdf

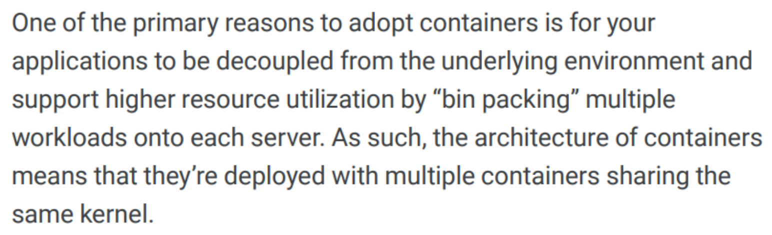
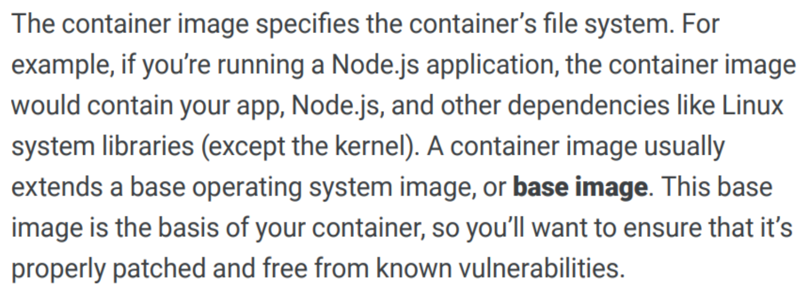
Claim 1	Accused Instrumentalities
	 Containers can run virtually anywhere, greatly easing development and deployment: on Linux, Windows, and Mac operating systems; on virtual machines or on physical servers; on a developer's machine or in data centers on-premises; and of course, in the public cloud. https://cloud.google.com/learn/what-are-containers

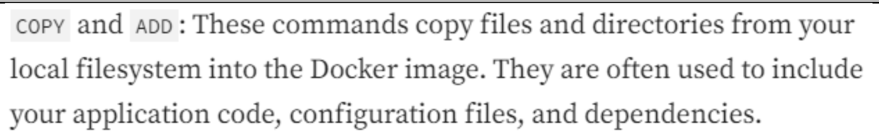
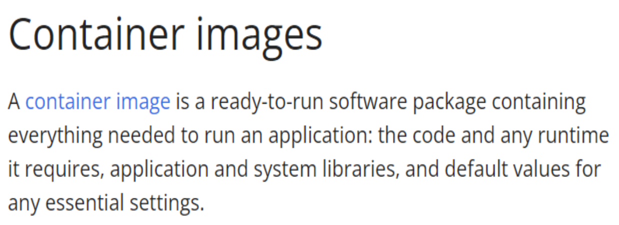
Claim 1	Accused Instrumentalities
[1c] the containers of application software excluding a kernel,	In the method practiced by Google and/or its customer through the Accused Instrumentalities, the containers of application software exclude a kernel. *See* discussion and evidence in element [1a] above. *See also, e.g.*:  • **Higher utilization and density**, leveraging automatic bin-packing and auto-scaling capabilities, Kubernetes places containers optimally in nodes based on required resources while scaling as needed, without impairing availability. In addition, unlike VMs, all containers on a single node share one copy of the operating system and don't each require their own OS image and vCPU, resulting in a much smaller memory footprint and CPU needs. This means more workloads running on fewer compute resources. https://cloud.google.com/blog/products/containers-kubernetes/how-migrate-for-anthos-improves-vm-to-container-migration  workloads onto each server. As such, the architecture of containers means that they're deployed with multiple containers sharing the same kernel.  Container A Container B host kernel virtual machine https://services.google.com/fh/files/misc/why_container_security_matters.pdf

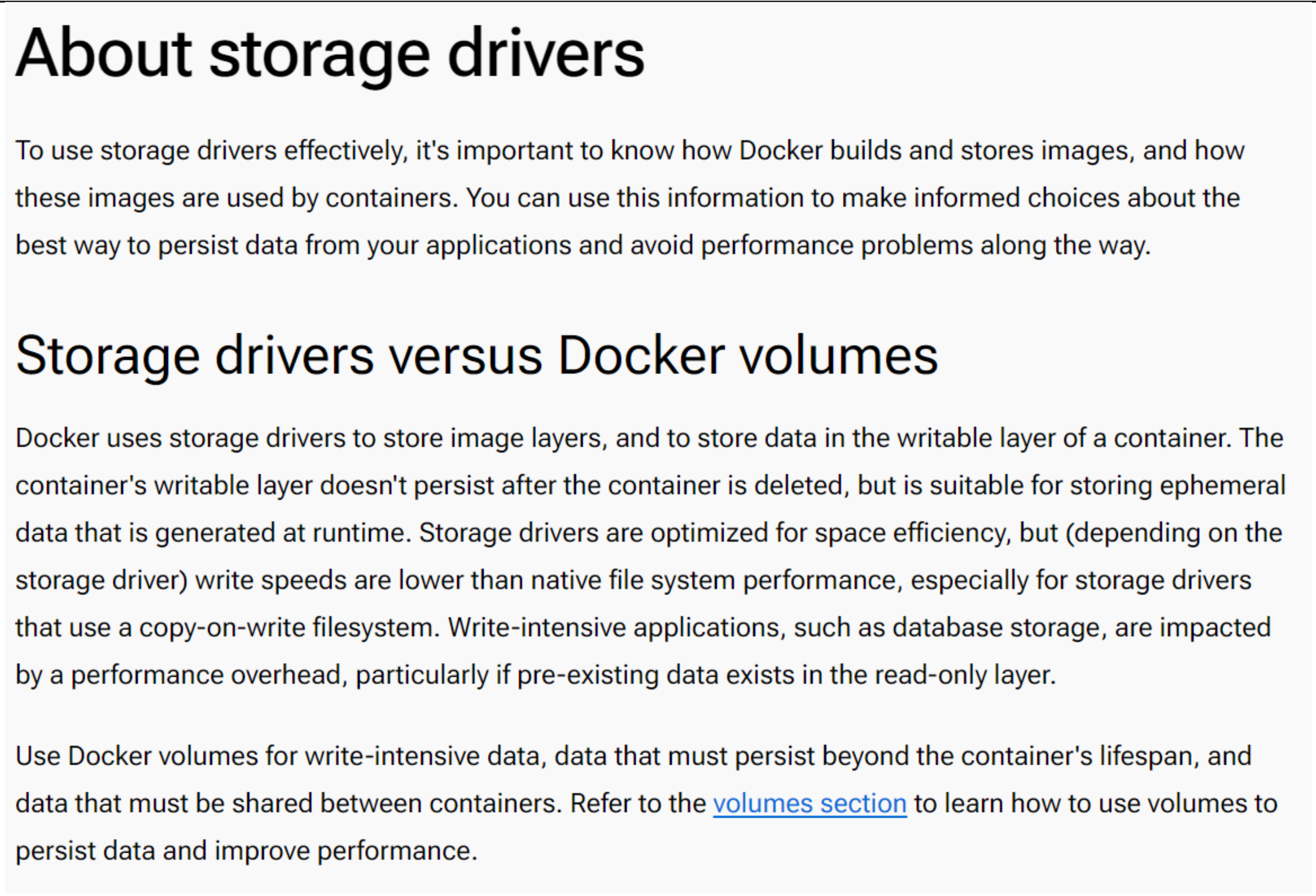
Claim 1	Accused Instrumentalities
	 6. **Do Docker containers package up the entire OS and make it easier to deploy?** Docker containers do not package up the OS. They package up the applications with everything that the application needs to run. The engine is installed on top of the OS running on a host. Containers share the OS kernel allowing a single host to run multiple containers. https://www.docker.com/blog/the-10-most-common-questions-it-admins-ask-about-docker/

Claim 1	Accused Instrumentalities
[1d] wherein some or all of the associated system files within a container stored in memory are utilized in place of the associated local system files that remain resident on the server,	In the method practiced by Google and/or its customer through the Accused Instrumentalities, some or all of the associated system files within a container stored in memory are utilized in place of the associated local system files that remain resident on the server. For example, each container will utilize its own ~~local~~ associated system files, including libraries such as libc/glibc and configuration files, not the corresponding associated local system files (*e.g.* libraries and configuration files of the host OS). As described above and below, in the Accused Instrumentalities the associated system files provide at least some of the same functionalities as the associated local system files. The host/node's associated local system files remain resident on the host/node, for example for use by system processes or applications outside the container environment. *See* discussion and evidence in element [1a] above. *See also, e.g.*: One of the primary reasons to adopt containers is for your applications to be decoupled from the underlying environment and support higher resource utilization by "bin packing" multiple workloads onto each server. As such, the architecture of containers means that they're deployed with multiple containers sharing the same kernel. The container image specifies the container's file system. For example, if you're running a Node.js application, the container image would contain your app, Node.js, and other dependencies like Linux system libraries (except the kernel). A container image usually extends a base operating system image, or **base image**. This base image is the basis of your container, so you'll want to ensure that it's properly patched and free from known vulnerabilities. https://services.google.com/fh/files/misc/why_container_security_matters.pdf

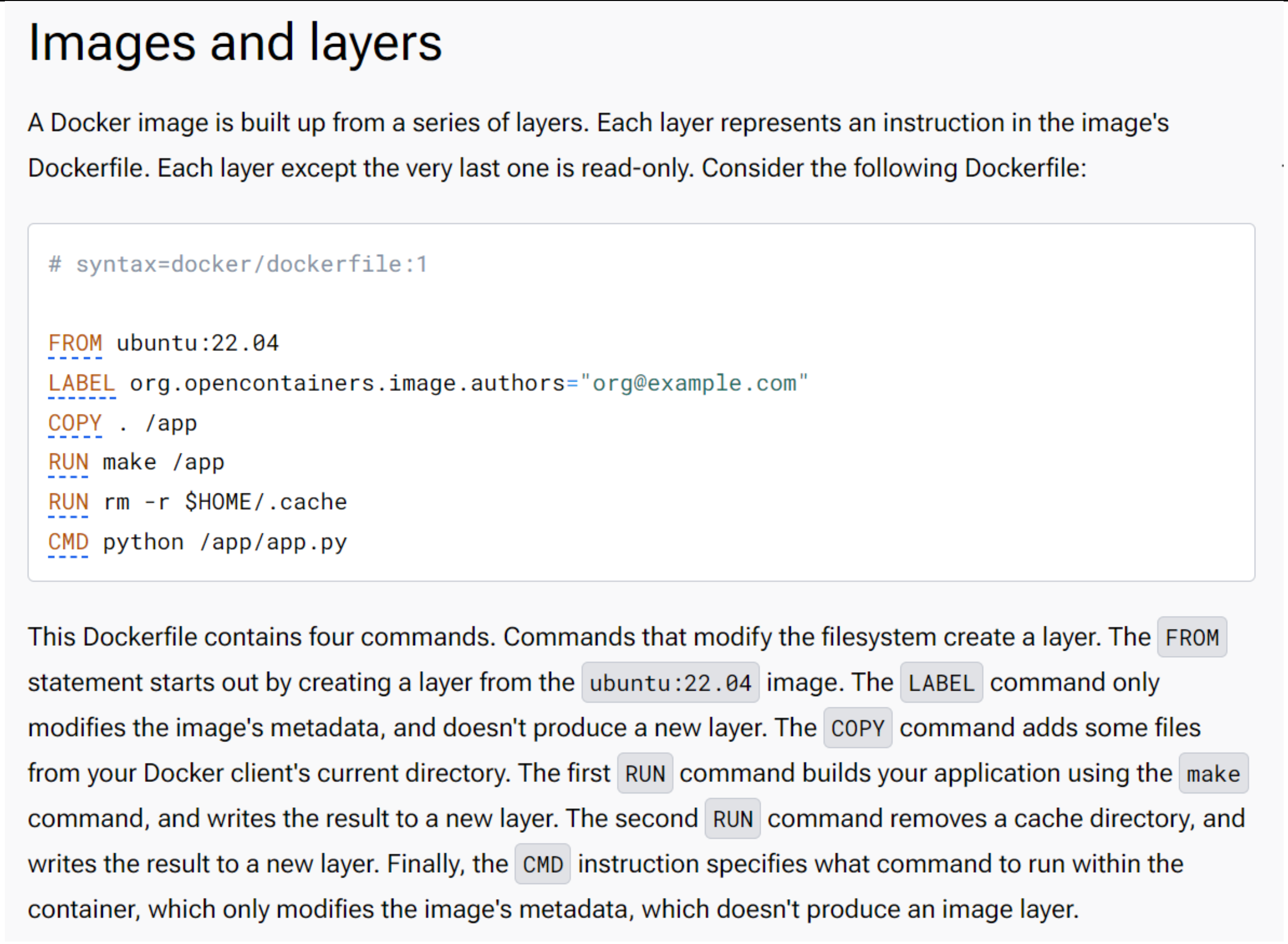
Claim 1	Accused Instrumentalities
	 Source Machine Services App 1 App 2 App 3 Other apps Logging Tomcat server OS Kernel + drivers File system Google containers platform Flexible deployment Container image App 2 Tomcat Services Persistent volume Networking Logging OS Kernel + drivers https://cloud.google.com/blog/products/application-modernization/shift-your-apps-to-container-based-workloads-on-the-command-line

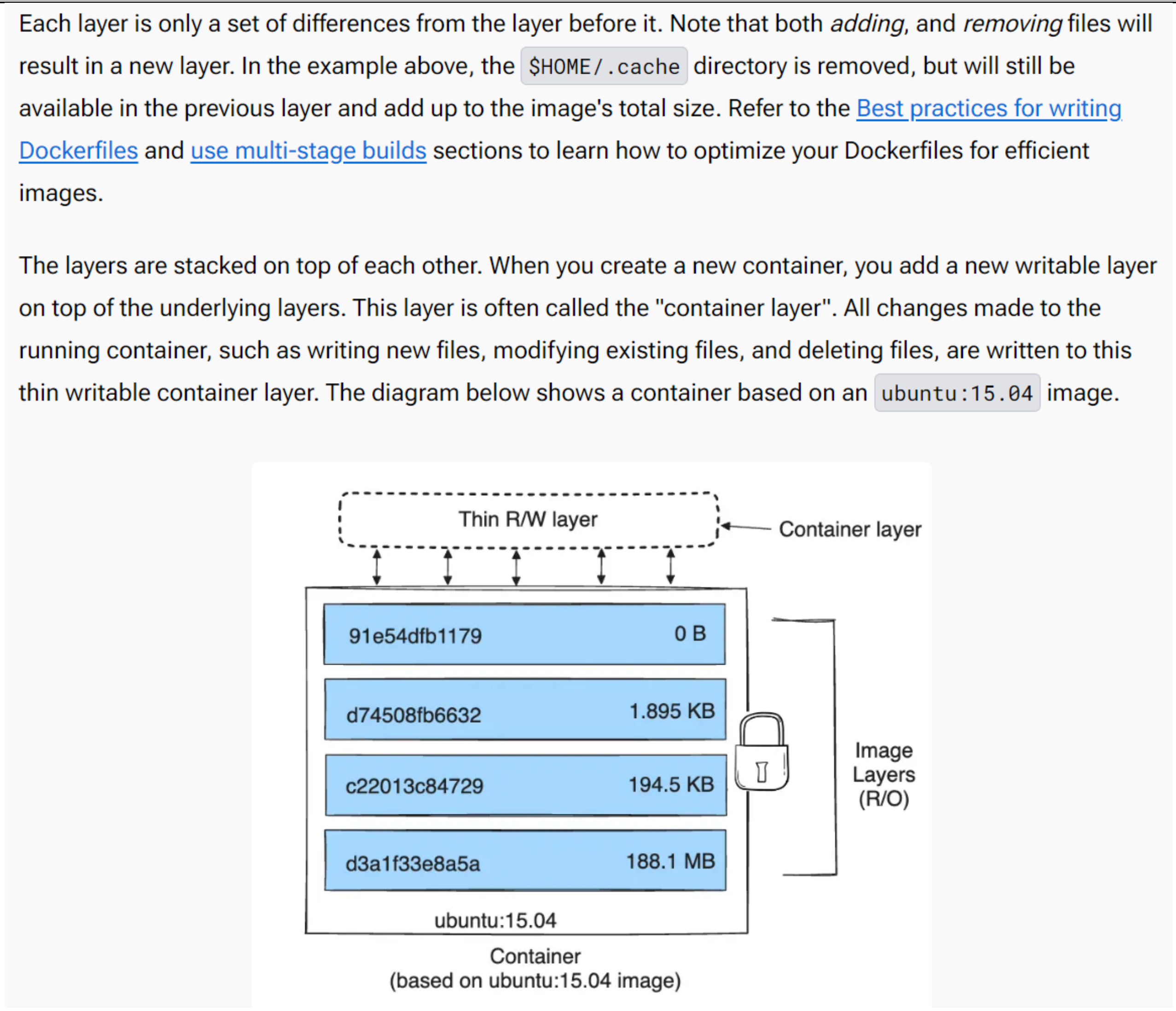
Claim 1	Accused Instrumentalities
[1e] wherein said associated system files utilized in place of the associated local system files are copies or modified copies of the associated local system files that remain resident on the server,	In the method practiced by Google and/or its customer through the Accused Instrumentalities, said associated system files utilized in place of the associated local system files are copies or modified copies of the associated local system files that remain resident on the server. For example, in some cases the host OS and container will use one or more identical system files, for example when both the host and the container incorporate the same Linux distribution version, or when both host and container use the same version of libc. In other cases modified copies are used instead, for example when different versions of the same library, or configuration files with different parameters, are used by the host and container. *See* discussion and evidence in element [1a] above. *See also, e.g.*:  One of the primary reasons to adopt containers is for your applications to be decoupled from the underlying environment and support higher resource utilization by "bin packing" multiple workloads onto each server. As such, the architecture of containers means that they're deployed with multiple containers sharing the same kernel.  The container image specifies the container's file system. For example, if you're running a Node.js application, the container image would contain your app, Node.js, and other dependencies like Linux system libraries (except the kernel). A container image usually extends a base operating system image, or **base image**. This base image is the basis of your container, so you'll want to ensure that it's properly patched and free from known vulnerabilities. https://services.google.com/fh/files/misc/why_container_security_matters.pdf

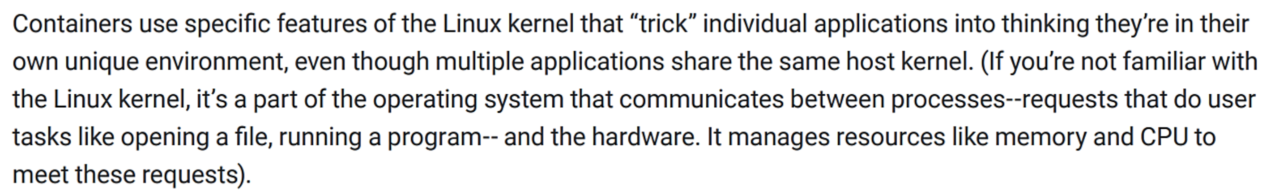
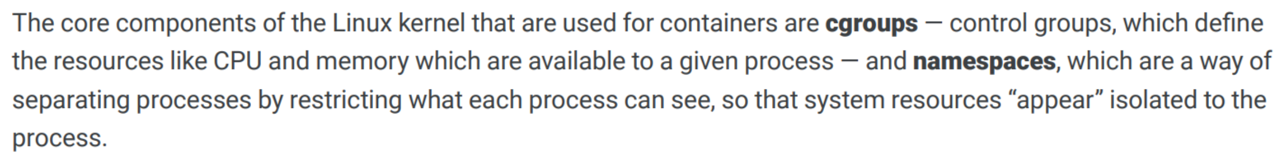
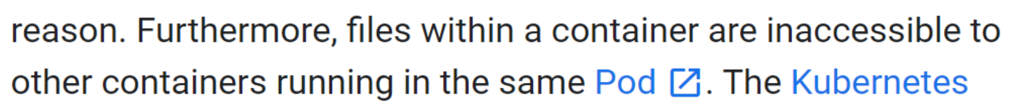
Claim 1	Accused Instrumentalities
	 `COPY` and `ADD`: These commands copy files and directories from your local filesystem into the Docker image. They are often used to include your application code, configuration files, and dependencies. https://medium.com/@swalperen3008/what-is-dockerize-and-dockerize-your-project-a-step-by-step-guide-899c48a34df6  Container images A container image is a ready-to-run software package containing everything needed to run an application: the code and any runtime it requires, application and system libraries, and default values for any essential settings. https://kubernetes.io/docs/concepts/containers/

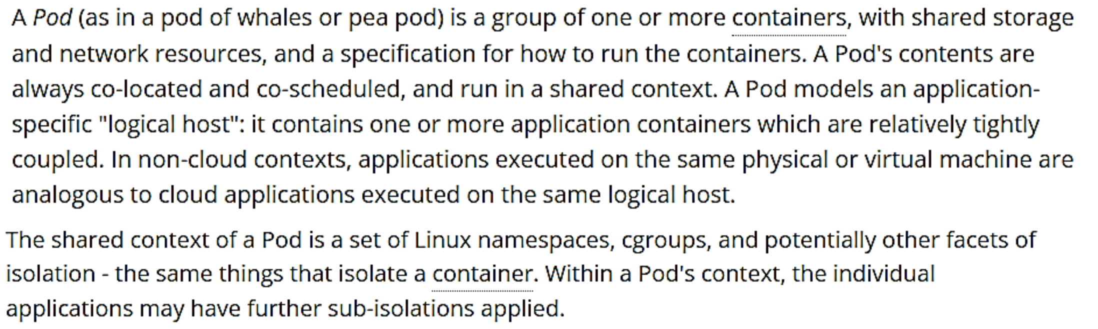
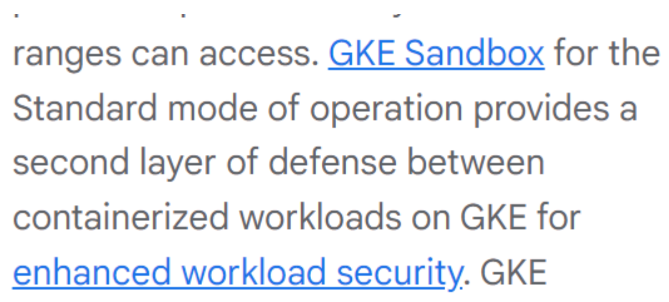
Claim 1	Accused Instrumentalities
	 **About storage drivers** To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. **Storage drivers versus Docker volumes** Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

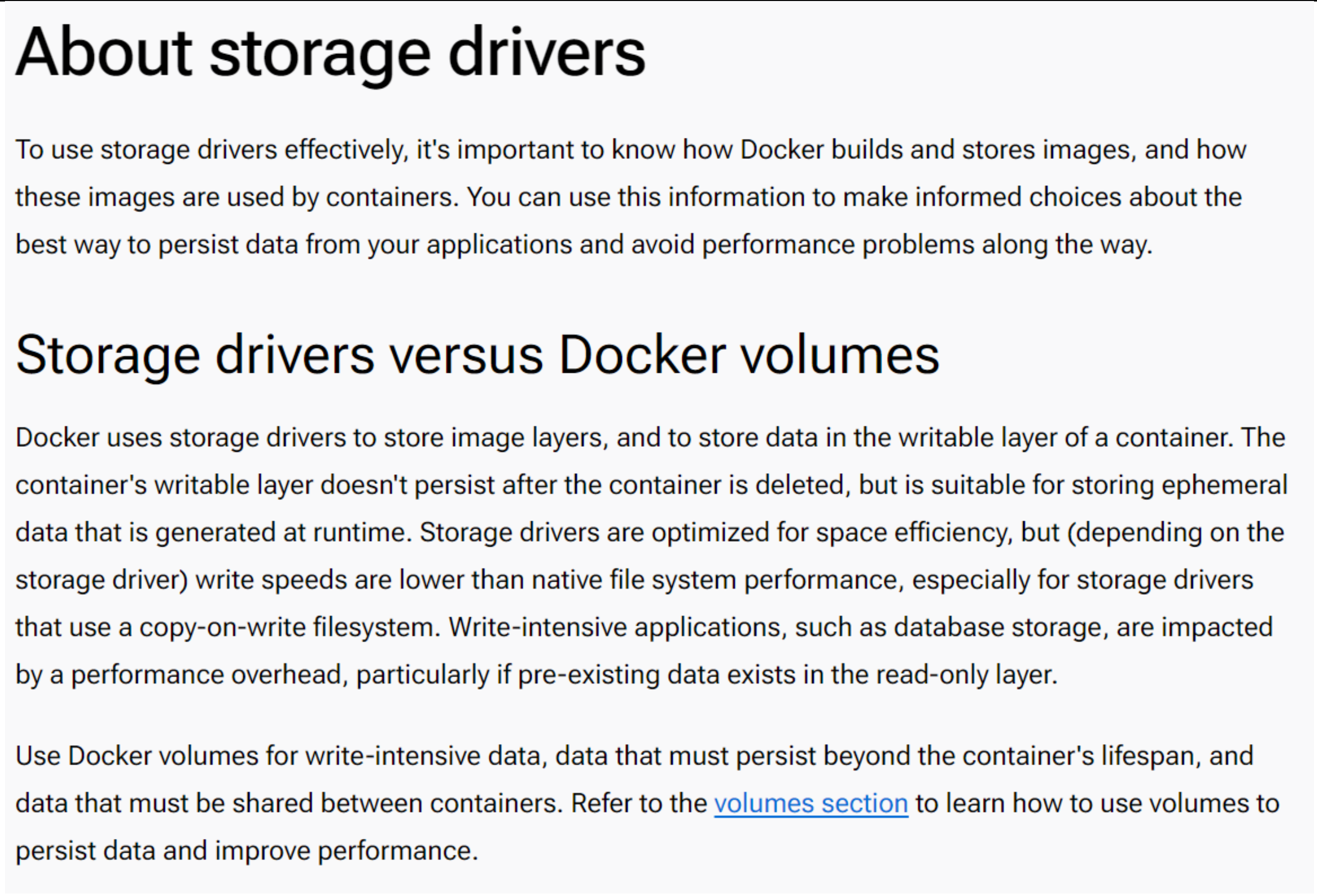
Claim 1	Accused Instrumentalities

	
	Images and layers
	A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:
	`# syntax=docker/dockerfile:1`
	
	`FROM ubuntu:22.04`
	`LABEL org.opencontainers.image.authors="org@example.com"`
	`COPY . /app`
	`RUN make /app`
	`RUN rm -r $HOME/.cache`
	`CMD python /app/app.py`
	
	This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.
	https://docs.docker.com/storage/storagedriver/

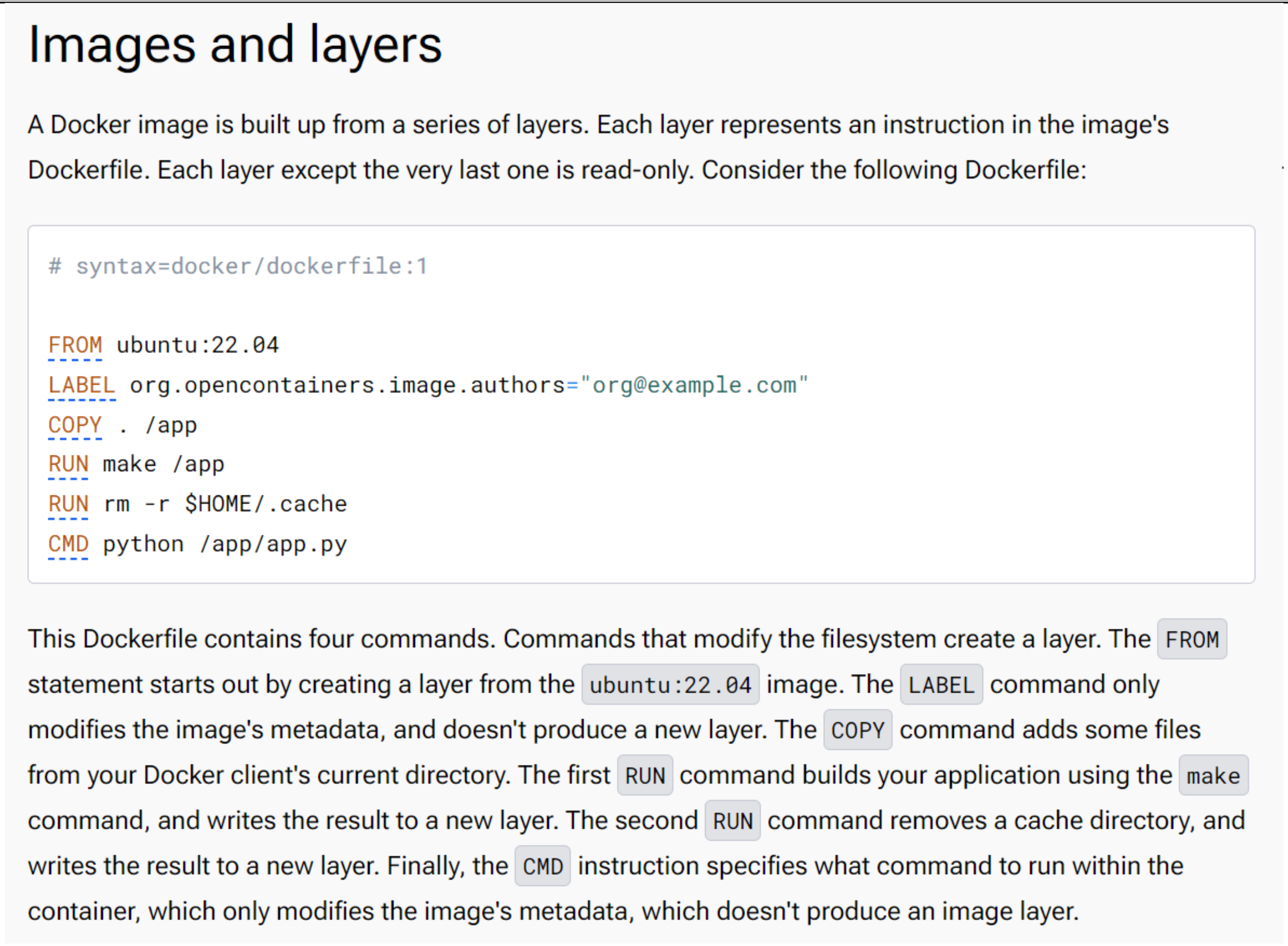
Claim 1	Accused Instrumentalities
	 Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image. Thin R/W layer Container layer 91e54dfb1179 0 B d74508fb6632 1.895 KB c22013c84729 194.5 KB d3a1f33e8a5a 188.1 MB Image Layers (R/O) ubuntu:15.04 Container (based on ubuntu:15.04 image) https://docs.docker.com/storage/storagedriver/

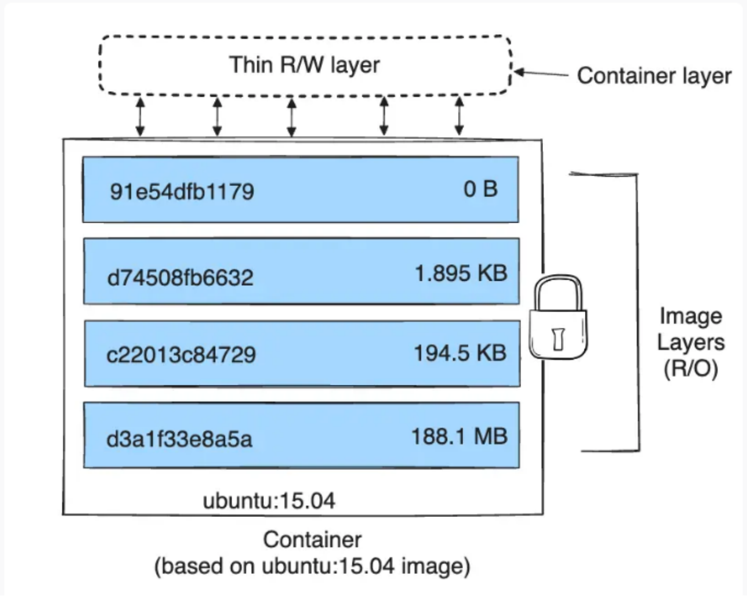
Claim 1	Accused Instrumentalities
[1f] and wherein the application software cannot be shared between the plurality of secure containers of application software,	In the method practiced by Google and/or its customer through the Accused Instrumentalities, the application software cannot be shared between the plurality of secure containers of application software. For example, each container has an isolated runtime environment that cannot be accessed by other containers, for example including a per-container writeable layer or other ephemeral per-container storage. For another example, when the plurality of secure containers each corresponds to a different container image, each container cannot access another container's image and therefore application software. *See, e.g.*:  Containers use specific features of the Linux kernel that "trick" individual applications into thinking they're in their own unique environment, even though multiple applications share the same host kernel. (If you're not familiar with the Linux kernel, it's a part of the operating system that communicates between processes--requests that do user tasks like opening a file, running a program-- and the hardware. It manages resources like memory and CPU to meet these requests). https://services.google.com/fh/files/misc/why_container_security_matters.pdf  The core components of the Linux kernel that are used for containers are **cgroups** — control groups, which define the resources like CPU and memory which are available to a given process — and **namespaces**, which are a way of separating processes by restricting what each process can see, so that system resources "appear" isolated to the process. https://services.google.com/fh/files/misc/why_container_security_matters.pdf  reason. Furthermore, files within a container are inaccessible to other containers running in the same Pod. The Kubernetes https://cloud.google.com/kubernetes-engine/docs/concepts/volumes

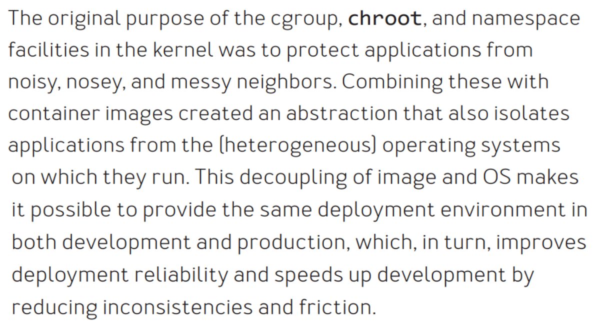
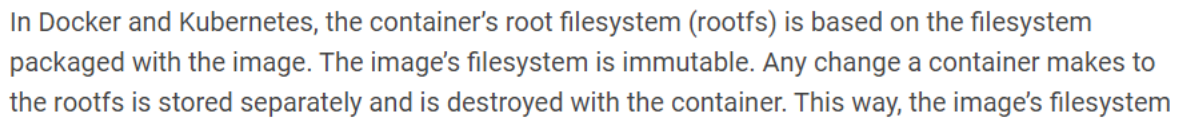
Claim 1	Accused Instrumentalities
	 A *Pod* (as in a pod of whales or pea pod) is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared context. A Pod models an application-specific "logical host": it contains one or more application containers which are relatively tightly coupled. In non-cloud contexts, applications executed on the same physical or virtual machine are analogous to cloud applications executed on the same logical host. The shared context of a Pod is a set of Linux namespaces, cgroups, and potentially other facets of isolation - the same things that isolate a container. Within a Pod's context, the individual applications may have further sub-isolations applied. https://kubernetes.io/docs/concepts/workloads/pods/  ranges can access. GKE Sandbox for the Standard mode of operation provides a second layer of defense between containerized workloads on GKE for enhanced workload security. GKE https://cloud.google.com/kubernetes-engine#section-2

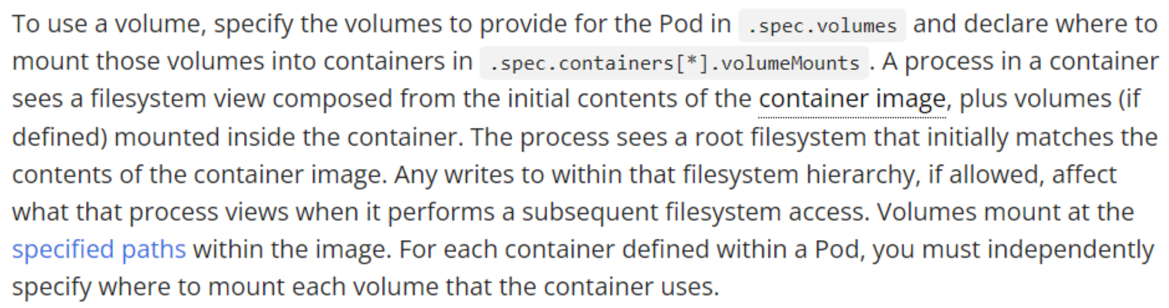
Claim 1	Accused Instrumentalities
	 About storage drivers To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. Storage drivers versus Docker volumes Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

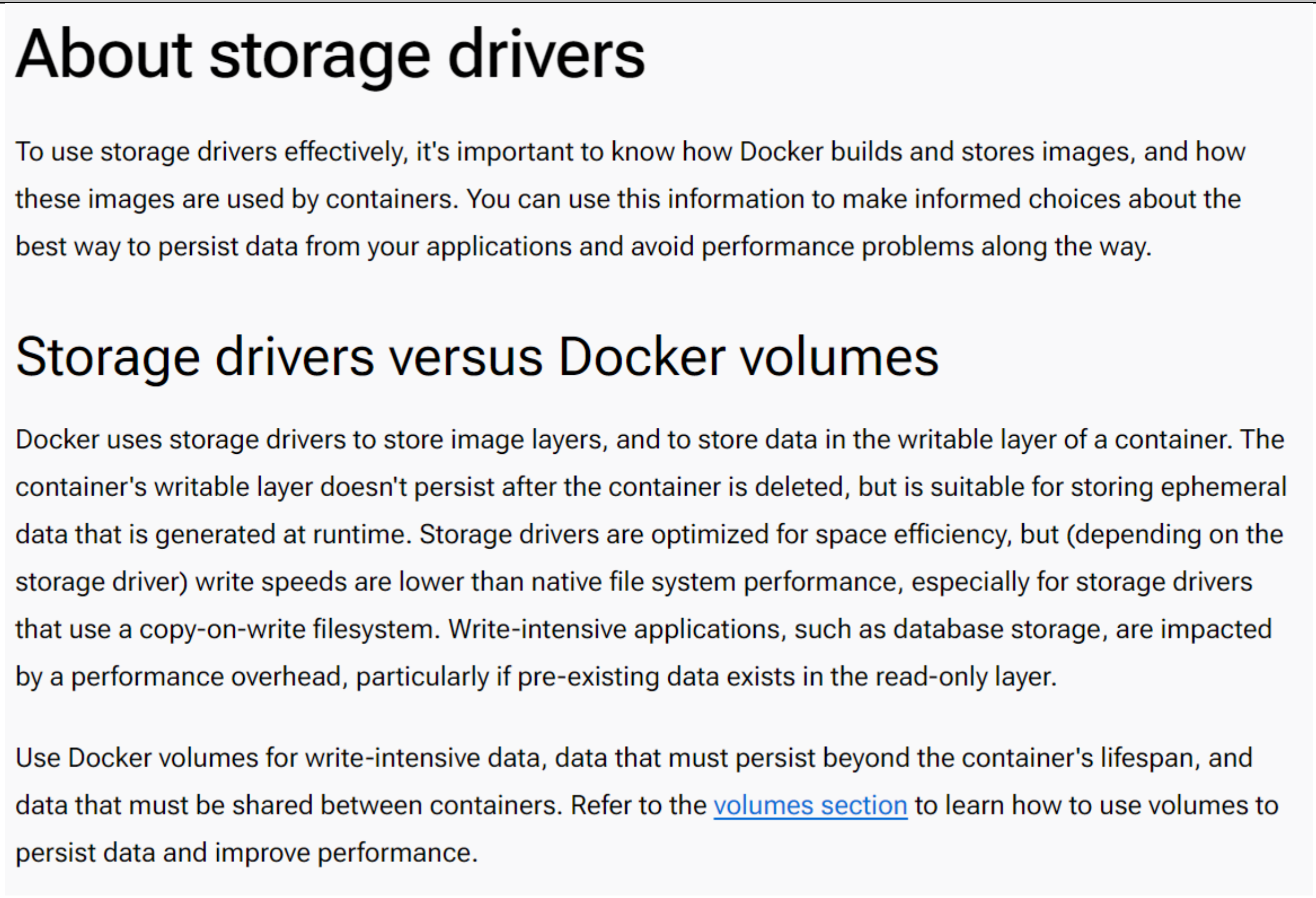
Claim 1	Accused Instrumentalities

	
	Images and layers
	A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:
	`# syntax=docker/dockerfile:1`
	
	`FROM ubuntu:22.04`
	`LABEL org.opencontainers.image.authors="org@example.com"`
	`COPY . /app`
	`RUN make /app`
	`RUN rm -r $HOME/.cache`
	`CMD python /app/app.py`
	This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.
	https://docs.docker.com/storage/storagedriver/

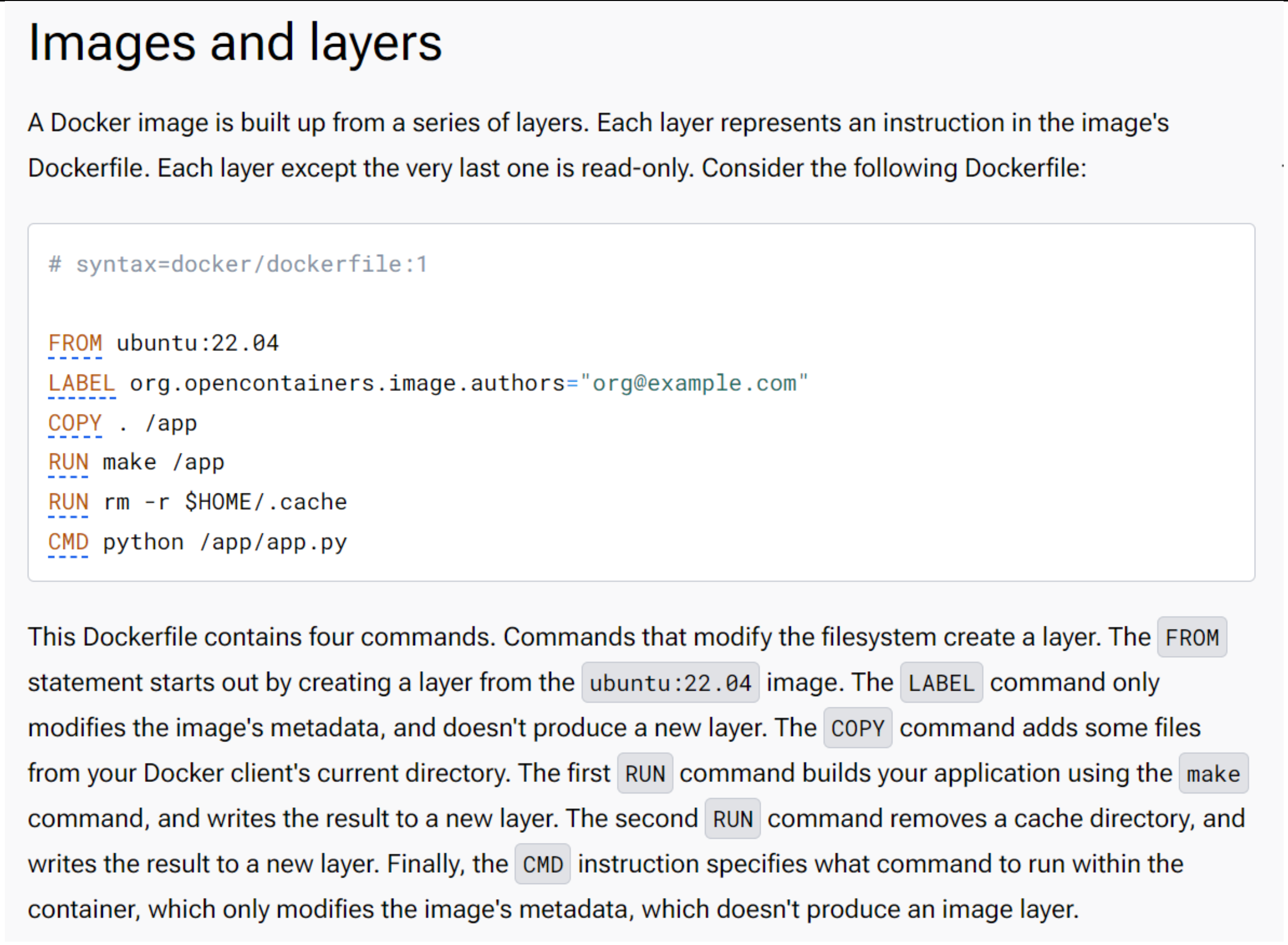
Claim 1	Accused Instrumentalities
	Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.  Thin R/W layer Container layer 91e54dfb1179 0 B d74508fb6632 1.895 KB c22013c84729 194.5 KB d3a1f33e8a5a 188.1 MB Image Layers (R/O) ubuntu:15.04 Container (based on ubuntu:15.04 image) https://docs.docker.com/storage/storagedriver/

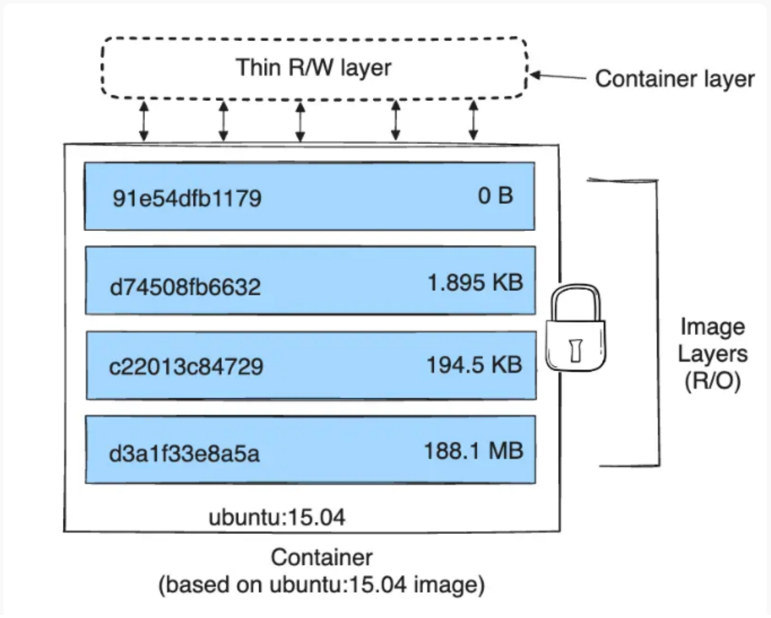
Claim 1	Accused Instrumentalities
[1g] and wherein each of the containers has a unique root file system that is different from an operating system's root file system.	In the method practiced by Google and/or its customer through the Accused Instrumentalities, each of the containers has a unique root file system that is different from an operating system's root file system. For example, the container's root file system comprises the image layer(s), an ephemeral writeable layer (e.g., in Docker terminology the container layer), and optionally one or more volumes. This root file system is distinct and isolated from the host operating system's root file system. *See, e.g.*: The original purpose of the cgroup, **`chroot`**, and namespace facilities in the kernel was to protect applications from noisy, nosey, and messy neighbors. Combining these with container images created an abstraction that also isolates applications from the (heterogeneous) operating systems on which they run. This decoupling of image and OS makes it possible to provide the same deployment environment in both development and production, which, in turn, improves deployment reliability and speeds up development by reducing inconsistencies and friction. "Borg, Omega, and, Kubernetes," https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/44843.pdf In Docker and Kubernetes, the container's root filesystem (rootfs) is based on the filesystem packaged with the image. The image's filesystem is immutable. Any change a container makes to the rootfs is stored separately and is destroyed with the container. This way, the image's filesystem https://opensource.googleblog.com/2023/04/gvisor-improves-performance-with-root-filesystem-overlay.html

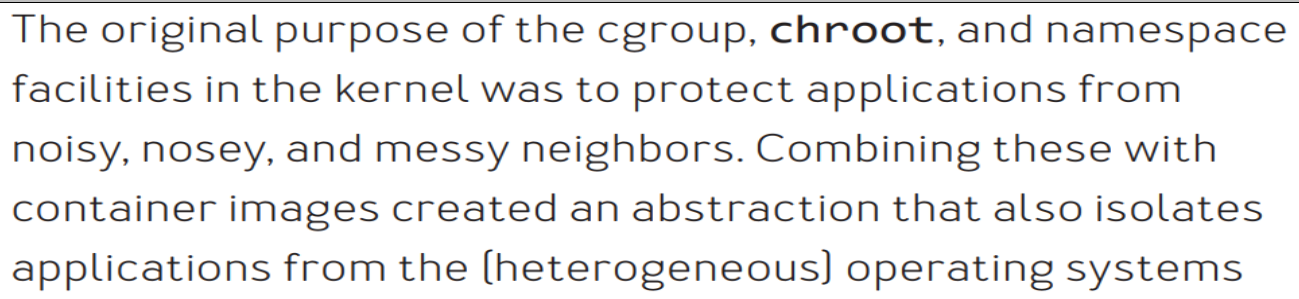
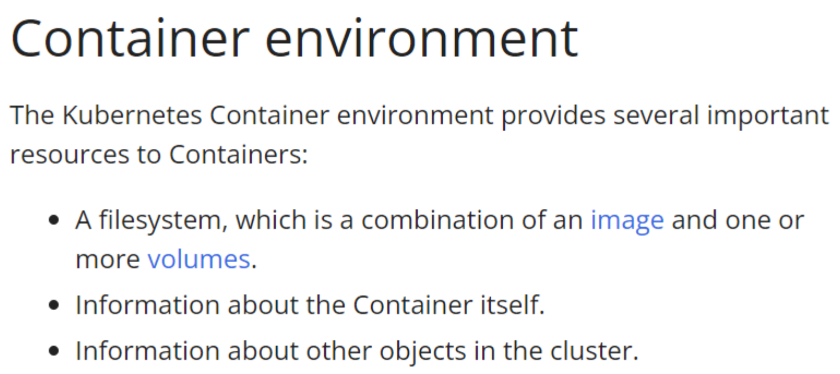
Claim 1	Accused Instrumentalities
	 To use a volume, specify the volumes to provide for the Pod in `.spec.volumes` and declare where to mount those volumes into containers in `.spec.containers[*].volumeMounts`. A process in a container sees a filesystem view composed from the initial contents of the container image, plus volumes (if defined) mounted inside the container. The process sees a root filesystem that initially matches the contents of the container image. Any writes to within that filesystem hierarchy, if allowed, affect what that process views when it performs a subsequent filesystem access. Volumes mount at the specified paths within the image. For each container defined within a Pod, you must independently specify where to mount each volume that the container uses. https://kubernetes.io/docs/concepts/storage/volumes/

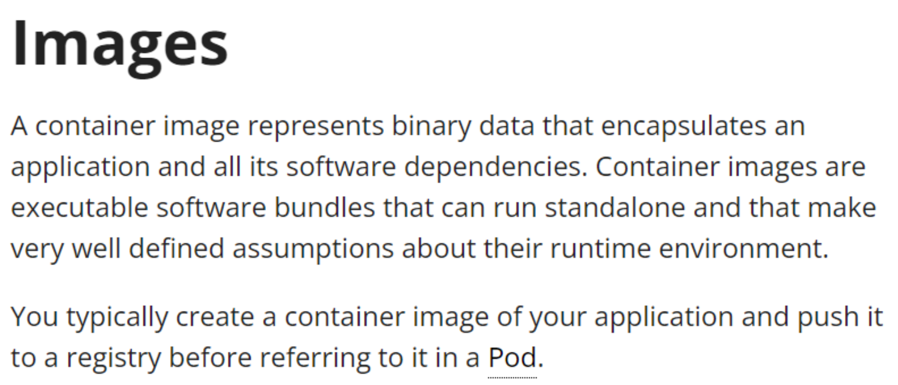
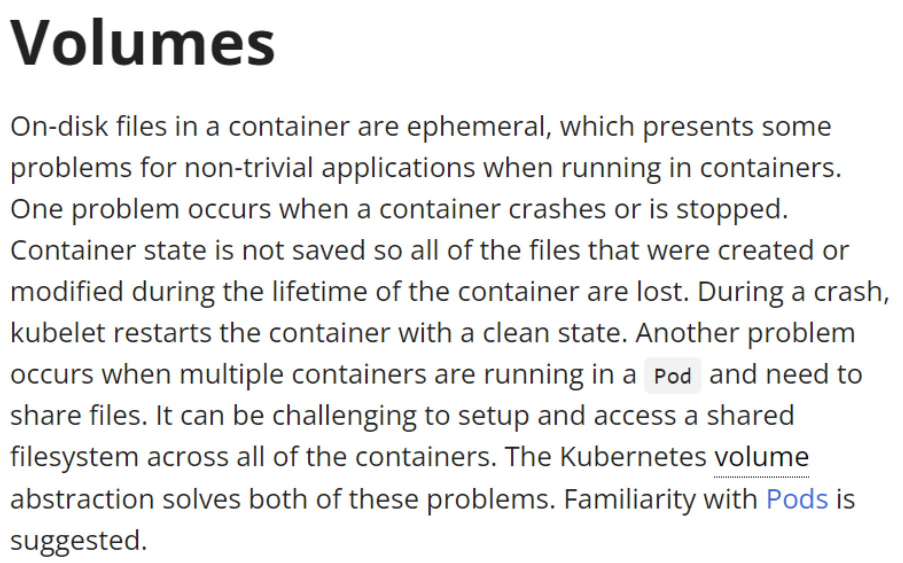
Claim 1	Accused Instrumentalities
	 ## About storage drivers To use storage drivers effectively, it's important to know how Docker builds and stores images, and how these images are used by containers. You can use this information to make informed choices about the best way to persist data from your applications and avoid performance problems along the way. ### Storage drivers versus Docker volumes Docker uses storage drivers to store image layers, and to store data in the writable layer of a container. The container's writable layer doesn't persist after the container is deleted, but is suitable for storing ephemeral data that is generated at runtime. Storage drivers are optimized for space efficiency, but (depending on the storage driver) write speeds are lower than native file system performance, especially for storage drivers that use a copy-on-write filesystem. Write-intensive applications, such as database storage, are impacted by a performance overhead, particularly if pre-existing data exists in the read-only layer. Use Docker volumes for write-intensive data, data that must persist beyond the container's lifespan, and data that must be shared between containers. Refer to the volumes section to learn how to use volumes to persist data and improve performance. https://docs.docker.com/storage/storagedriver/

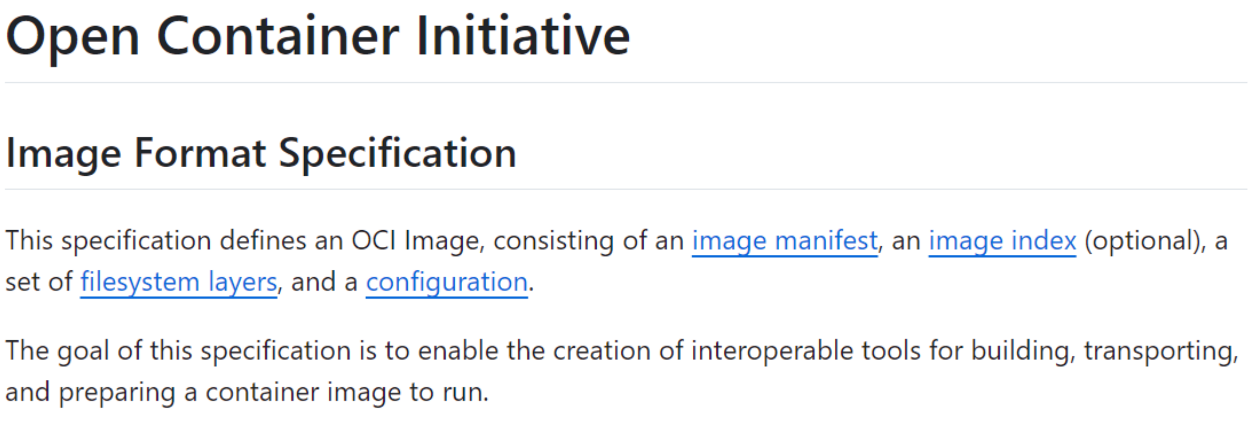
Claim 1	Accused Instrumentalities

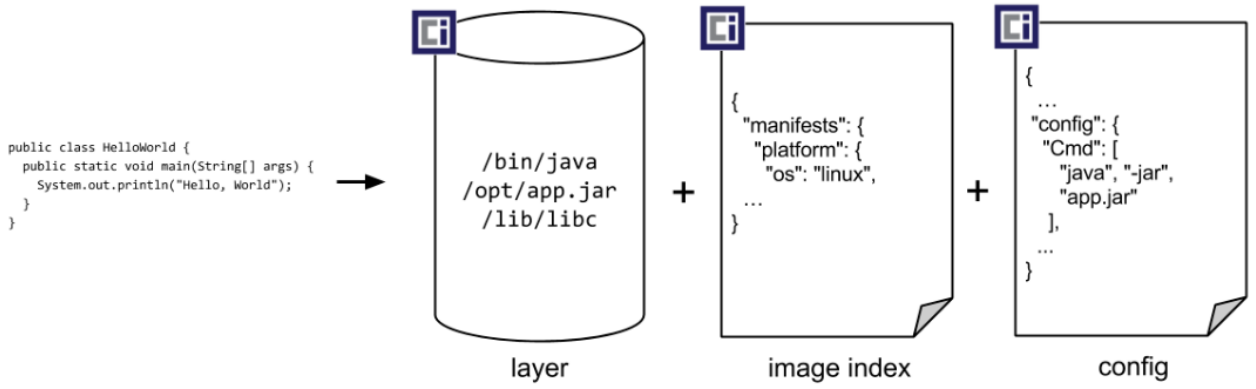
	 ## Images and layers
	
	A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile:
	
	`# syntax=docker/dockerfile:1`
	
	`FROM ubuntu:22.04`
	`LABEL org.opencontainers.image.authors="org@example.com"`
	`COPY . /app`
	`RUN make /app`
	`RUN rm -r $HOME/.cache`
	`CMD python /app/app.py`
	
	This Dockerfile contains four commands. Commands that modify the filesystem create a layer. The `FROM` statement starts out by creating a layer from the `ubuntu:22.04` image. The `LABEL` command only modifies the image's metadata, and doesn't produce a new layer. The `COPY` command adds some files from your Docker client's current directory. The first `RUN` command builds your application using the `make` command, and writes the result to a new layer. The second `RUN` command removes a cache directory, and writes the result to a new layer. Finally, the `CMD` instruction specifies what command to run within the container, which only modifies the image's metadata, which doesn't produce an image layer.
	
	https://docs.docker.com/storage/storagedriver/

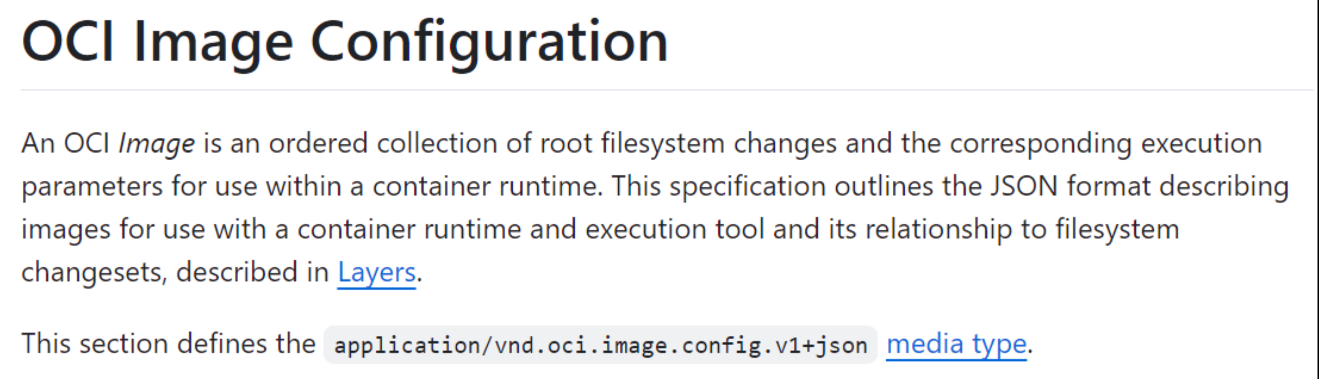
Claim 1	Accused Instrumentalities
	Each layer is only a set of differences from the layer before it. Note that both *adding*, and *removing* files will result in a new layer. In the example above, the `$HOME/.cache` directory is removed, but will still be available in the previous layer and add up to the image's total size. Refer to the Best practices for writing Dockerfiles and use multi-stage builds sections to learn how to optimize your Dockerfiles for efficient images. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. The diagram below shows a container based on an `ubuntu:15.04` image.  Thin R/W layer Container layer 91e54dfb1179 0 B d74508fb6632 1.895 KB c22013c84729 194.5 KB d3a1f33e8a5a 188.1 MB ubuntu:15.04 Image Layers (R/O) Container (based on ubuntu:15.04 image) https://docs.docker.com/storage/storagedriver/

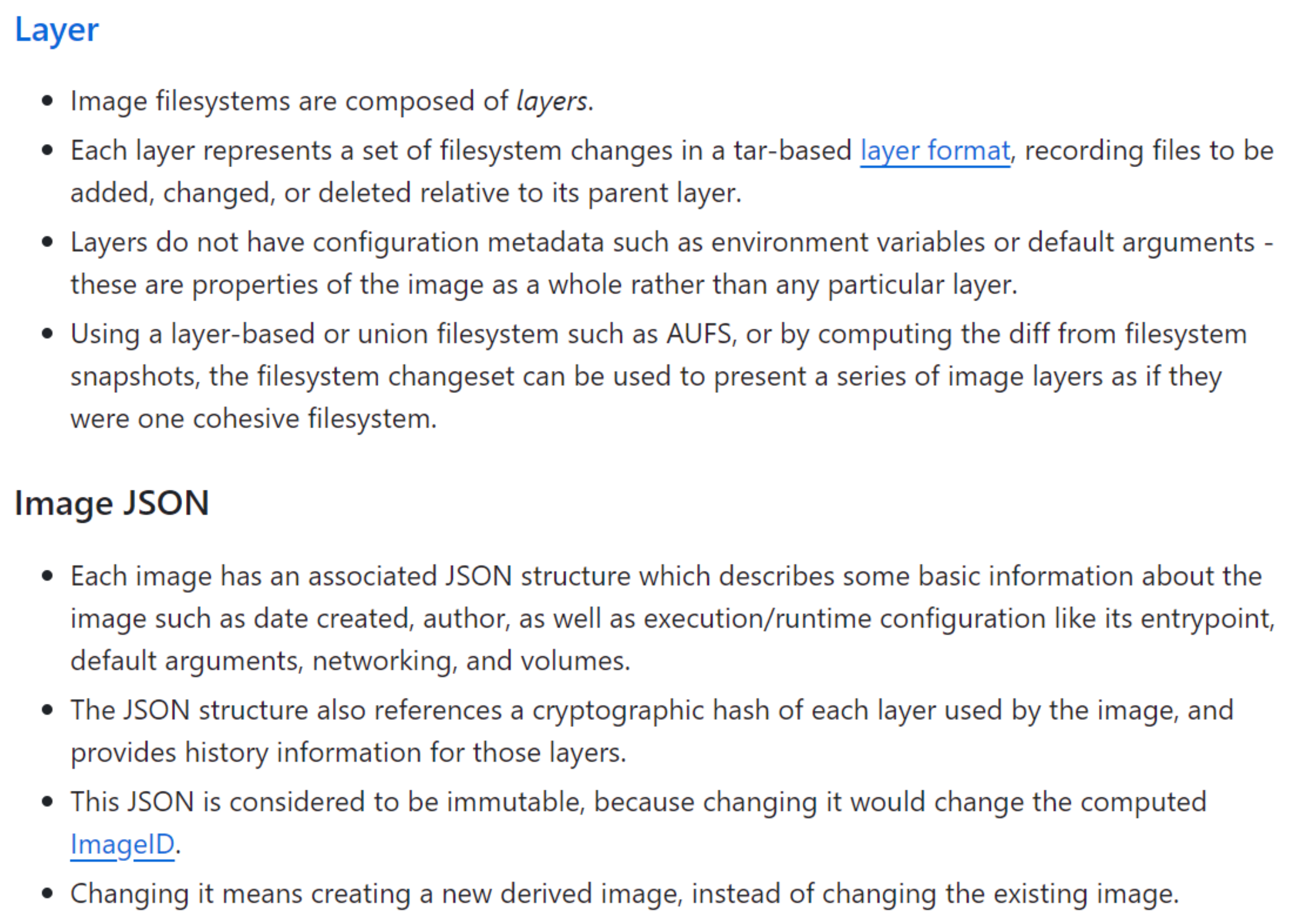
Claim 1	Accused Instrumentalities
	 The original purpose of the cgroup, **chroot**, and namespace facilities in the kernel was to protect applications from noisy, nosey, and messy neighbors. Combining these with container images created an abstraction that also isolates applications from the (heterogeneous) operating systems https://kubernetes.io/docs/concepts/storage/volumes/  **Container environment** The Kubernetes Container environment provides several important resources to Containers: • A filesystem, which is a combination of an image and one or more volumes. • Information about the Container itself. • Information about other objects in the cluster. https://kubernetes.io/docs/concepts/containers/container-environment/

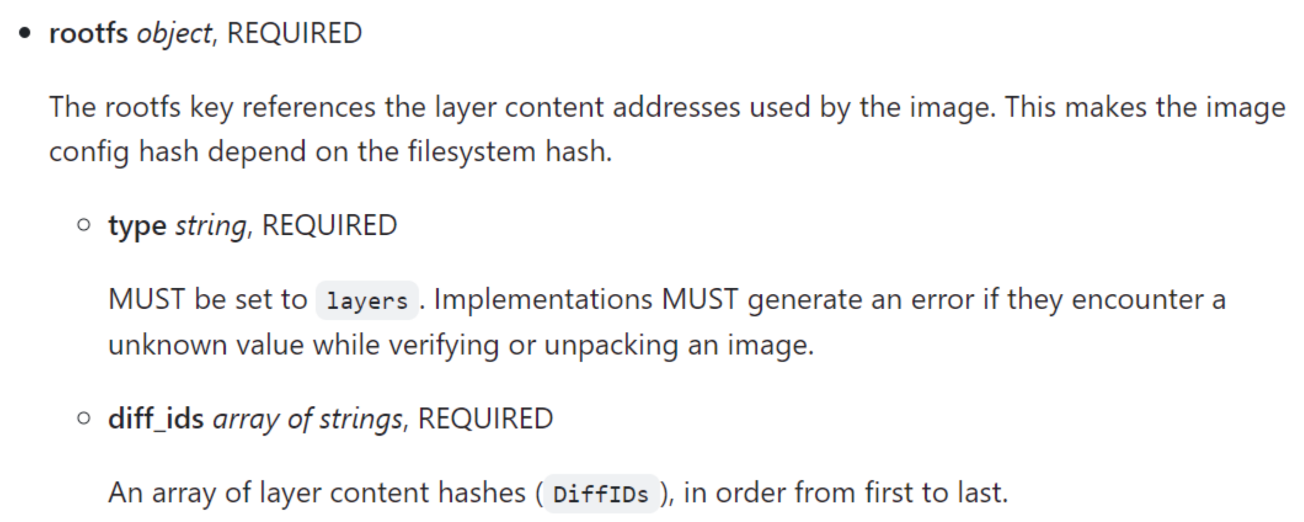
Claim 1	Accused Instrumentalities
	 **Images** A container image represents binary data that encapsulates an application and all its software dependencies. Container images are executable software bundles that can run standalone and that make very well defined assumptions about their runtime environment. You typically create a container image of your application and push it to a registry before referring to it in a Pod. https://kubernetes.io/docs/concepts/containers/images/  **Volumes** On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem occurs when a container crashes or is stopped. Container state is not saved so all of the files that were created or modified during the lifetime of the container are lost. During a crash, kubelet restarts the container with a clean state. Another problem occurs when multiple containers are running in a `Pod` and need to share files. It can be challenging to setup and access a shared filesystem across all of the containers. The Kubernetes volume abstraction solves both of these problems. Familiarity with Pods is suggested. https://kubernetes.io/docs/concepts/storage/volumes/

Claim 1	Accused Instrumentalities
	 Open Container Initiative Image Format Specification This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

Claim 1	Accused Instrumentalities
	Overview At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.  public class HelloWorld { public static void main(String[] args) { System.out.println("Hello, World"); } } /bin/java /opt/app.jar /lib/libc layer { "manifests": { "platform": { "os": "linux", ... } image index { ... "config": { "Cmd": ["java", "-jar", "app.jar"], ... } config https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

Claim 1	Accused Instrumentalities
	 **OCI Image Configuration** An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

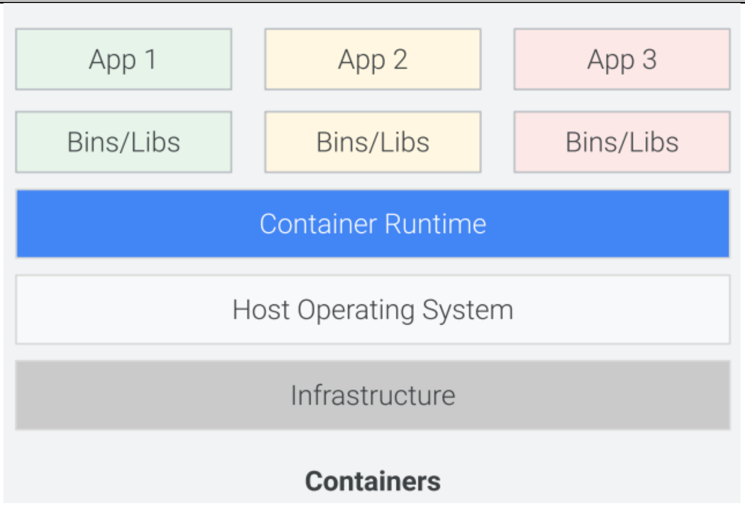
Claim 1	Accused Instrumentalities
	 **Layer** • Image filesystems are composed of *layers*. • Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer. • Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer. • Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem. **Image JSON** • Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes. • The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers. • This JSON is considered to be immutable, because changing it would change the computed ImageID. • Changing it means creating a new derived image, instead of changing the existing image. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

Claim 1	Accused Instrumentalities
	 • **rootfs** *object*, REQUIRED The rootfs key references the layer content addresses used by the image. This makes the image config hash depend on the filesystem hash. ○ **type** *string*, REQUIRED MUST be set to `layers`. Implementations MUST generate an error if they encounter a unknown value while verifying or unpacking an image. ○ **diff_ids** *array of strings*, REQUIRED An array of layer content hashes (`DiffIDs`), in order from first to last. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

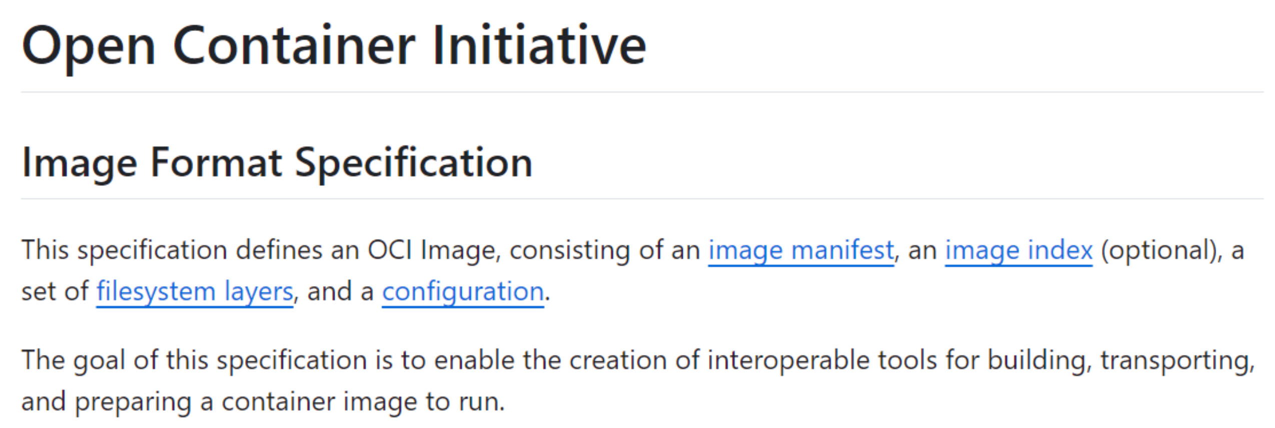
Claim 2

Claim 2	Accused Instrumentalities
2. A method as defined in claim 1, wherein each container has an execution file associated therewith for starting the one or more applications.	Google and/or its customer practices, through the Accused Instrumentalities, a method as defined in claim 1, wherein each container has an execution file associated therewith for starting the one or more applications. For example, a container image has an associated image configuration comprising information for starting the one or more applications. This can be an Open Containers Initiative image configuration. *See, e.g.*:

<table>
<tr><th>Claim 2</th><th>Accused Instrumentalities</th></tr>
<tr><td></td><td>



https://services.google.com/fh/files/misc/why_container_security_matters.pdf



Open Container Initiative

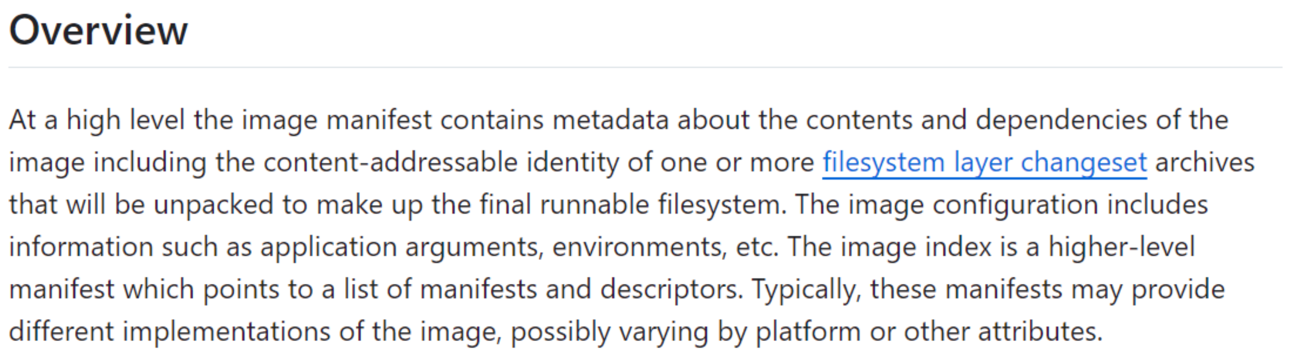
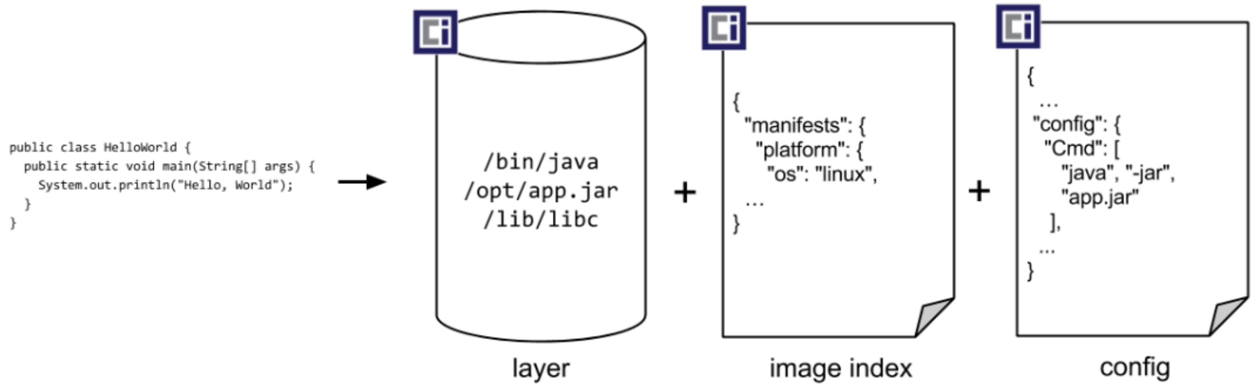
Image Format Specification

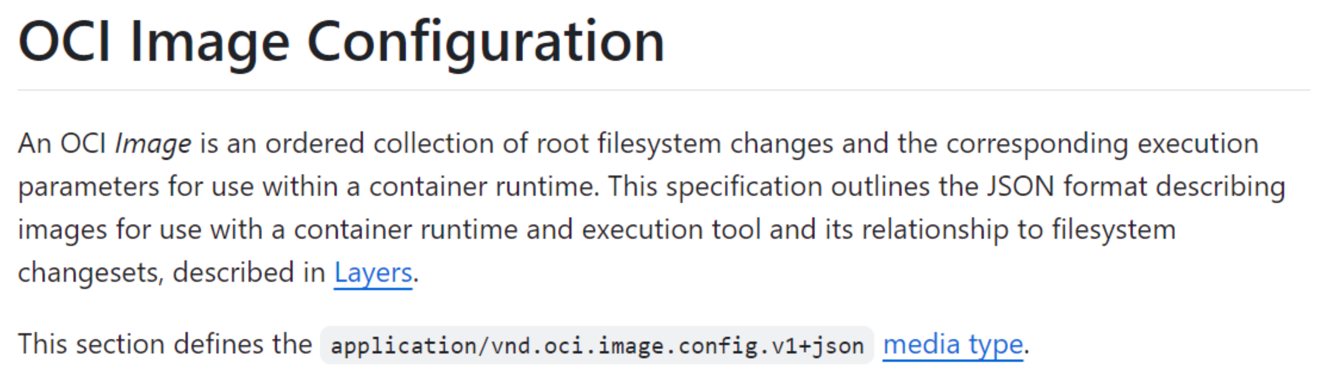
This specification defines an OCI Image, consisting of an image manifest, an image index (optional), a set of filesystem layers, and a configuration.

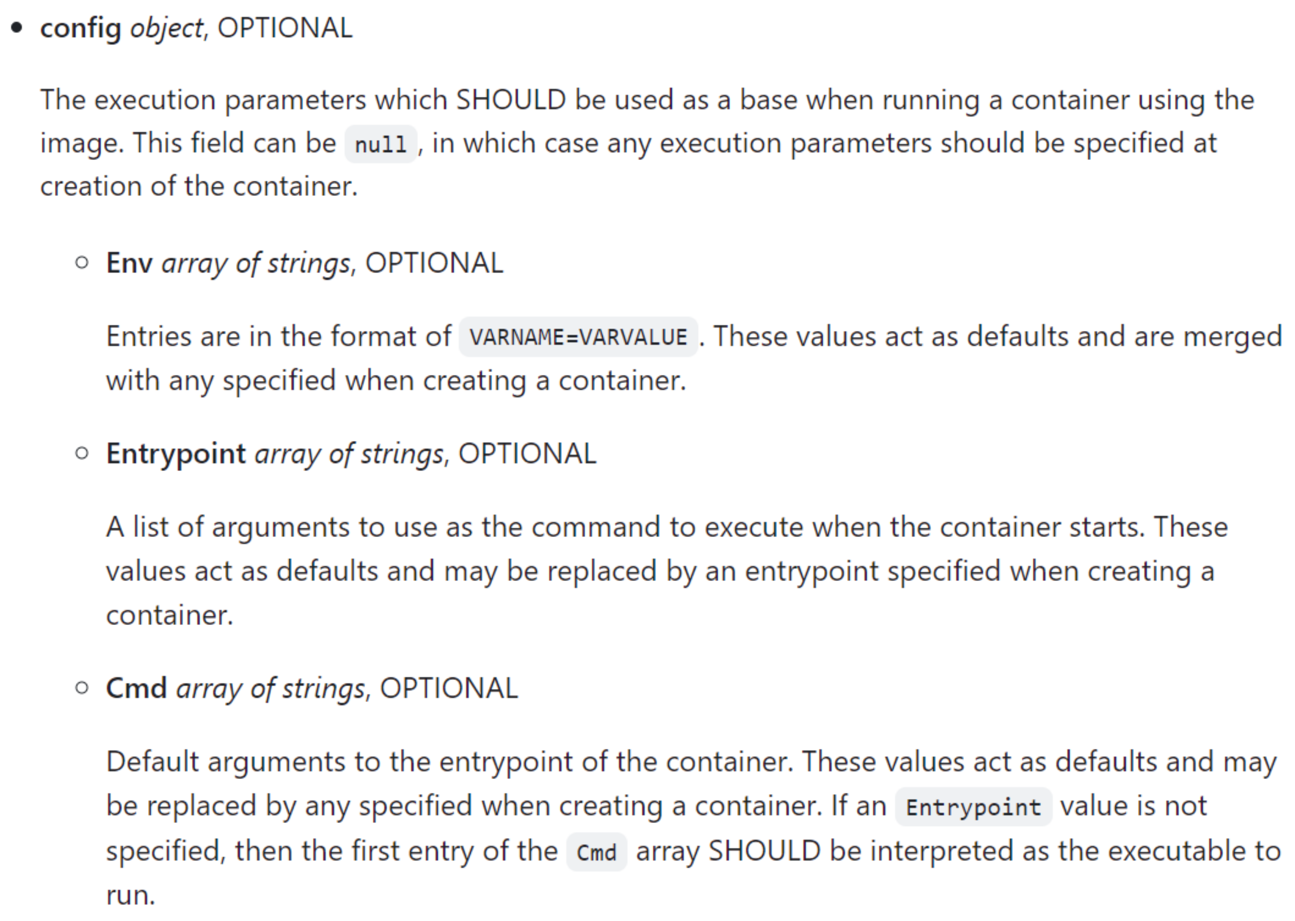
The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run.

https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

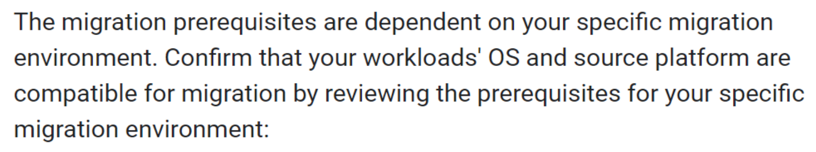
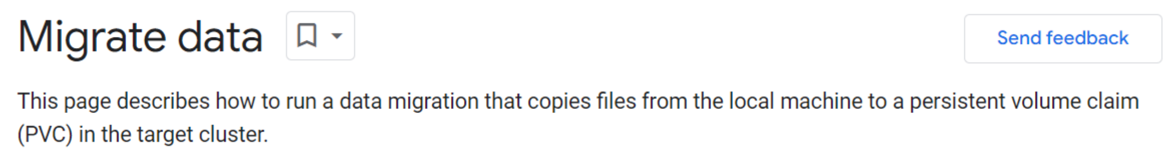
</td></tr>
</table>

Claim 2	Accused Instrumentalities
	 **Overview** At a high level the image manifest contains metadata about the contents and dependencies of the image including the content-addressable identity of one or more filesystem layer changeset archives that will be unpacked to make up the final runnable filesystem. The image configuration includes information such as application arguments, environments, etc. The image index is a higher-level manifest which points to a list of manifests and descriptors. Typically, these manifests may provide different implementations of the image, possibly varying by platform or other attributes.  public class HelloWorld { public static void main(String[] args) { System.out.println("Hello, World"); } } /bin/java /opt/app.jar /lib/libc { "manifests": { "platform": { "os": "linux", … } { … "config": { "Cmd": ["java", "-jar", "app.jar"], … } layer image index config https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/spec.md

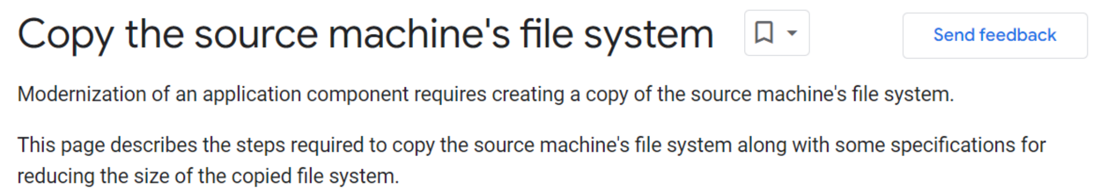
Claim 2	Accused Instrumentalities
	 OCI Image Configuration An OCI *Image* is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers. This section defines the `application/vnd.oci.image.config.v1+json` media type. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

Claim 2	Accused Instrumentalities
	 • **config** *object*, OPTIONAL The execution parameters which SHOULD be used as a base when running a container using the image. This field can be `null`, in which case any execution parameters should be specified at creation of the container. ◦ **Env** *array of strings*, OPTIONAL Entries are in the format of `VARNAME=VARVALUE`. These values act as defaults and are merged with any specified when creating a container. ◦ **Entrypoint** *array of strings*, OPTIONAL A list of arguments to use as the command to execute when the container starts. These values act as defaults and may be replaced by an entrypoint specified when creating a container. ◦ **Cmd** *array of strings*, OPTIONAL Default arguments to the entrypoint of the container. These values act as defaults and may be replaced by any specified when creating a container. If an `Entrypoint` value is not specified, then the first entry of the `Cmd` array SHOULD be interpreted as the executable to run. https://github.com/opencontainers/image-spec/blob/a6af2b480dcfc001ba975f44de53001c873cb0ef/config.md

Claim 4

Claim 4	Accused Instrumentalities
4. A method as defined in claim 1 further comprising the step of pre-identifying applications and system files required for association with the one or more containers prior to said storing step.	Google and/or its customer practices, through the Accused Instrumentalities, a method as defined in claim 1 further comprising the step of pre-identifying applications and system files required for association with the one or more containers prior to said storing step. For example, in the infringing configuration where Google's Migrate to Containers is used to produce a container that is run on Google Kubernetes Engine or Cloud Run, ~~Google's~~ Migrate to Containers ~~feature~~ identifies the application along with its dependencies to be migrated to the target cluster/container. This identification step happens before storing the containers having the migrated application and files in the target machine, as described above. *See* analysis and evidence for claim 1 above. *See also, e.g.*:  The migration prerequisites are dependent on your specific migration environment. Confirm that your workloads' OS and source platform are compatible for migration by reviewing the prerequisites for your specific migration environment: https://cloud.google.com/migrate/containers/docs/setting-up-overview  Migrate data This page describes how to run a data migration that copies files from the local machine to a persistent volume claim (PVC) in the target cluster. https://cloud.google.com/migrate/containers/docs/m2c-cli/migrate-data

Claim 4	Accused Instrumentalities



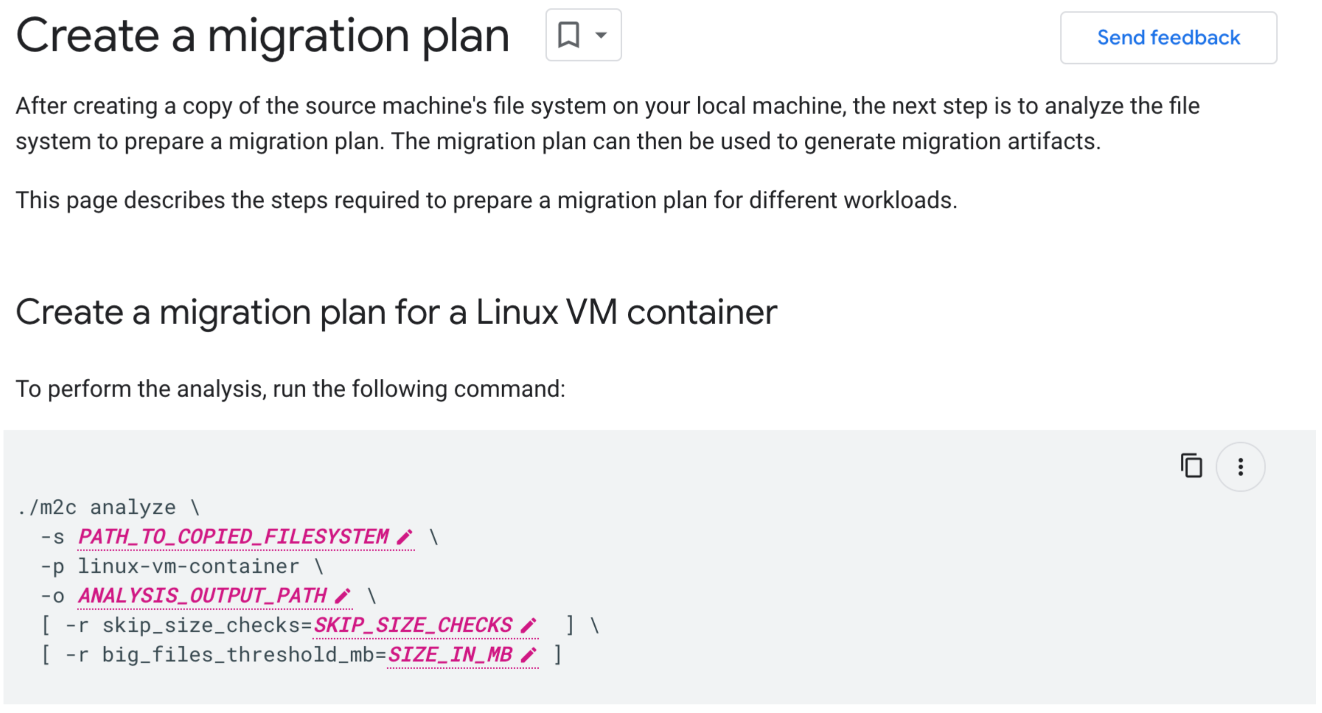
Copy the source machine's file system

Send feedback

Modernization of an application component requires creating a copy of the source machine's file system.

This page describes the steps required to copy the source machine's file system along with some specifications for reducing the size of the copied file system.

https://cloud.google.com/migrate/containers/docs/m2c-cli/copy-file-system



Create a migration plan

Send feedback

After creating a copy of the source machine's file system on your local machine, the next step is to analyze the file system to prepare a migration plan. The migration plan can then be used to generate migration artifacts.

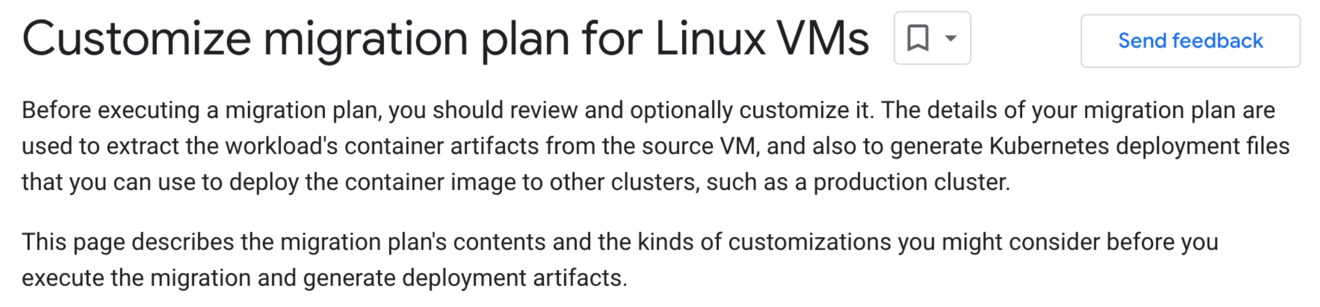
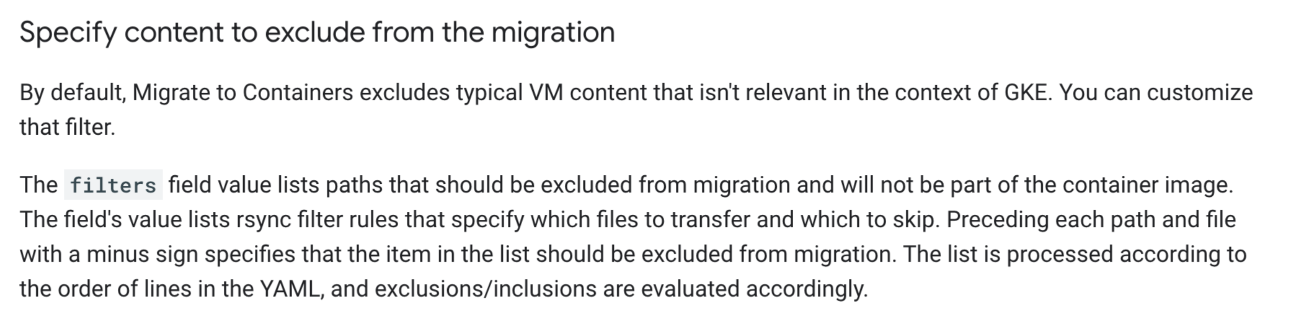
This page describes the steps required to prepare a migration plan for different workloads.

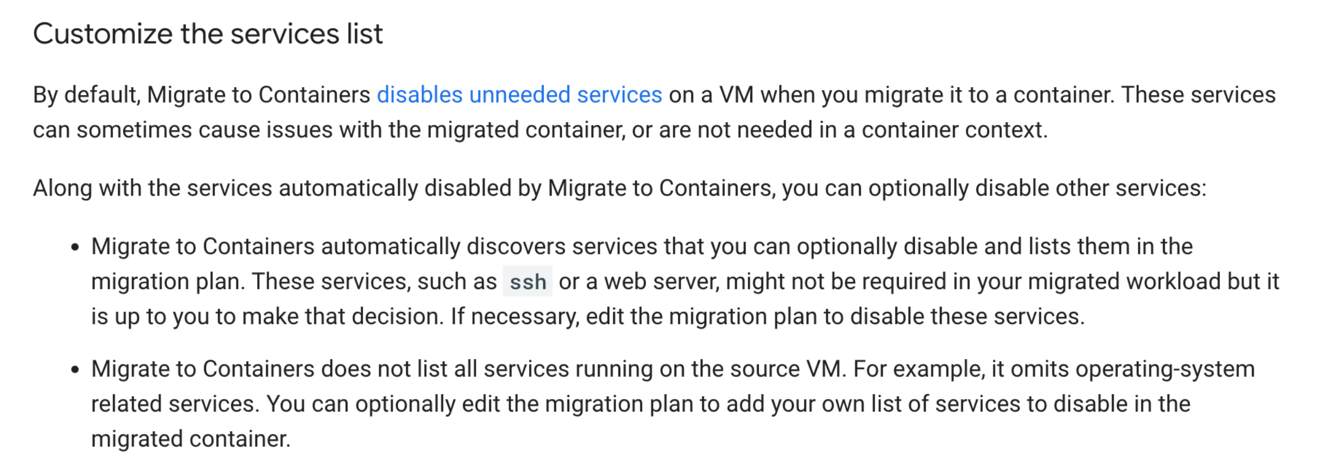
Create a migration plan for a Linux VM container

To perform the analysis, run the following command:

```
./m2c analyze \
  -s PATH_TO_COPIED_FILESYSTEM \
  -p linux-vm-container \
  -o ANALYSIS_OUTPUT_PATH \
  [ -r skip_size_checks=SKIP_SIZE_CHECKS  ] \
  [ -r big_files_threshold_mb=SIZE_IN_MB  ]
```

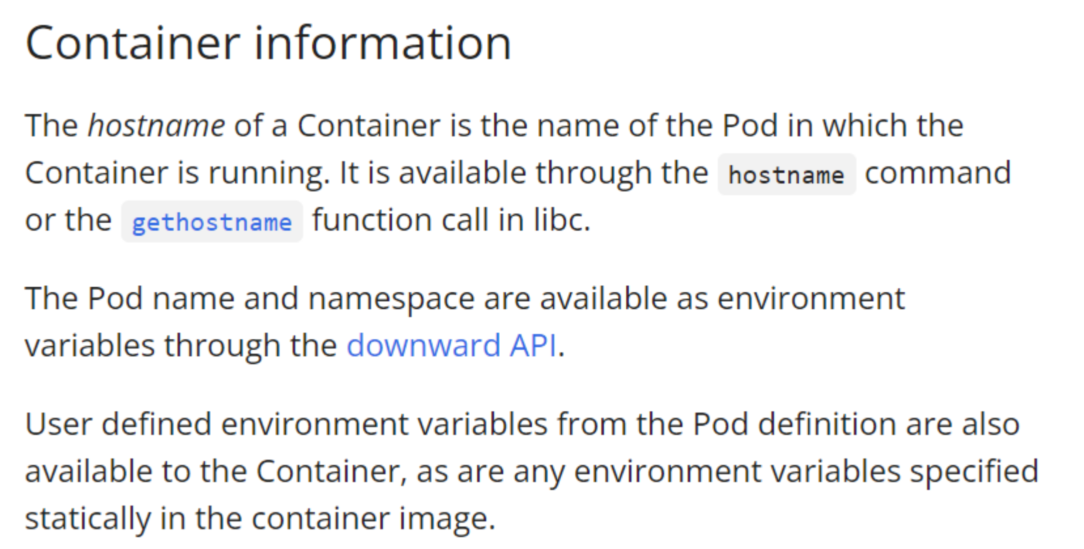
https://cloud.google.com/migrate/containers/docs/m2c-cli/create-a-migration-plan

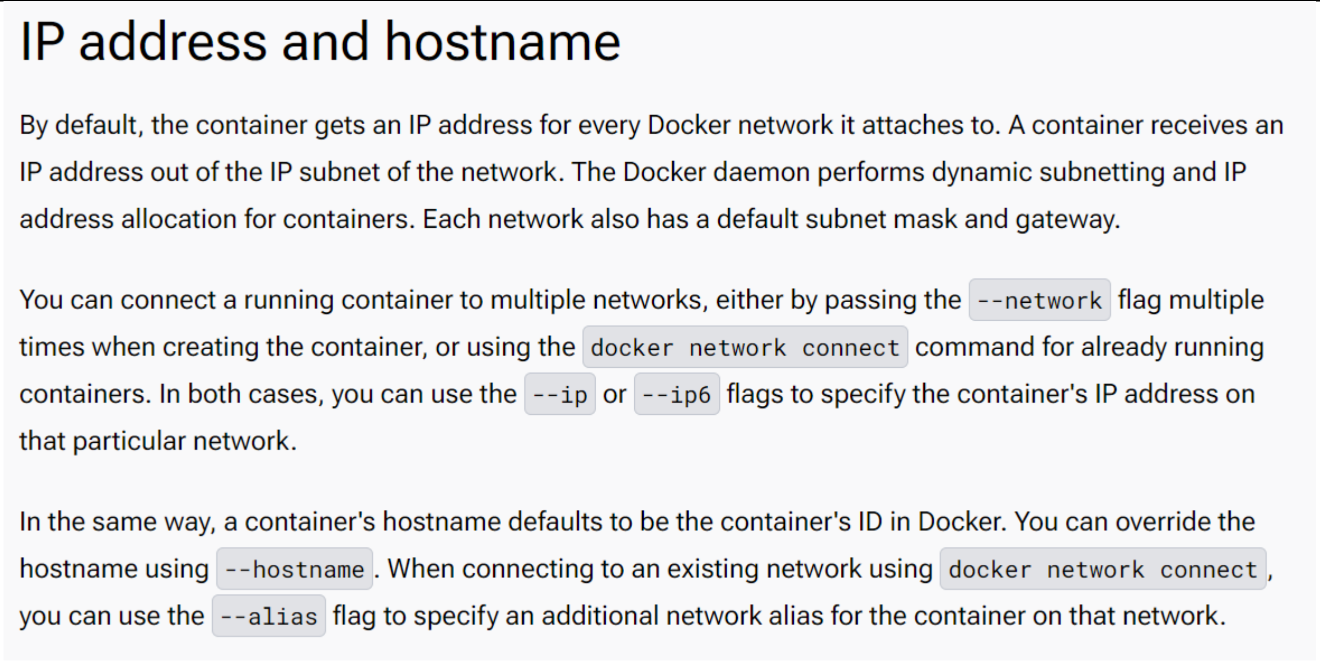
Claim 4	Accused Instrumentalities
	 Customize migration plan for Linux VMs Send feedback Before executing a migration plan, you should review and optionally customize it. The details of your migration plan are used to extract the workload's container artifacts from the source VM, and also to generate Kubernetes deployment files that you can use to deploy the container image to other clusters, such as a production cluster. This page describes the migration plan's contents and the kinds of customizations you might consider before you execute the migration and generate deployment artifacts. https://cloud.google.com/migrate/containers/docs/m2c-cli/linux/customizing-a-migration-plan  Specify content to exclude from the migration By default, Migrate to Containers excludes typical VM content that isn't relevant in the context of GKE. You can customize that filter. The `filters` field value lists paths that should be excluded from migration and will not be part of the container image. The field's value lists rsync filter rules that specify which files to transfer and which to skip. Preceding each path and file with a minus sign specifies that the item in the list should be excluded from migration. The list is processed according to the order of lines in the YAML, and exclusions/inclusions are evaluated accordingly. https://cloud.google.com/migrate/containers/docs/m2c-cli/linux/customizing-a-migration-plan

Claim 4	Accused Instrumentalities
	 Customize the services list By default, Migrate to Containers disables unneeded services on a VM when you migrate it to a container. These services can sometimes cause issues with the migrated container, or are not needed in a container context. Along with the services automatically disabled by Migrate to Containers, you can optionally disable other services: • Migrate to Containers automatically discovers services that you can optionally disable and lists them in the migration plan. These services, such as `ssh` or a web server, might not be required in your migrated workload but it is up to you to make that decision. If necessary, edit the migration plan to disable these services. • Migrate to Containers does not list all services running on the source VM. For example, it omits operating-system related services. You can optionally edit the migration plan to add your own list of services to disable in the migrated container. https://cloud.google.com/migrate/containers/docs/m2c-cli/linux/customizing-a-migration-plan

Claim 6

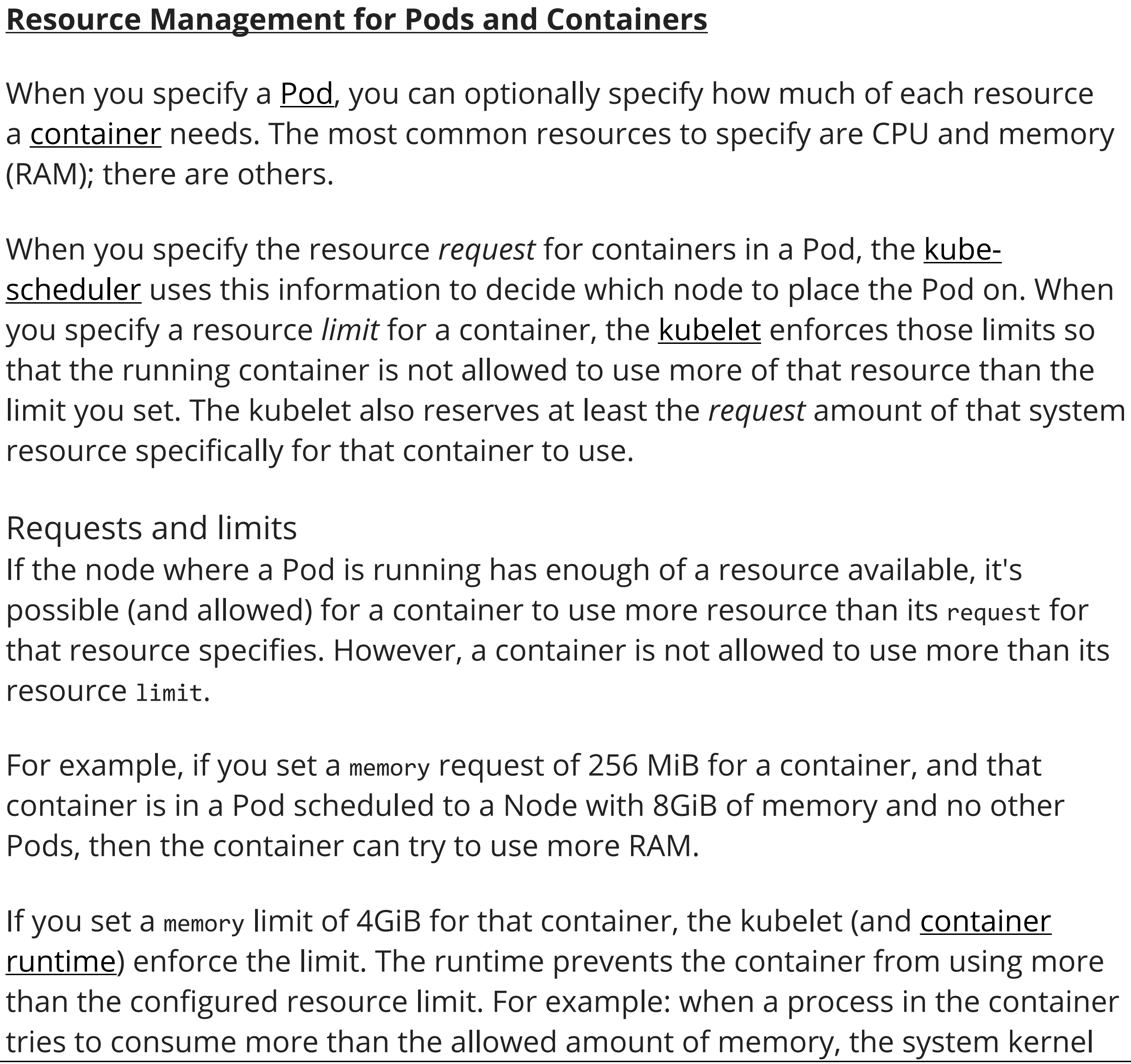
Claim 6	Accused Instrumentalities
6. A method as defined in claim 2, comprising the step of assigning a unique associated identity to each of a plurality of the containers, wherein the identity includes at least one of IP address, host name, and MAC address.	Google and/or its customer practices, through the Accused Instrumentalities, a method as defined in claim 2, comprising the step of assigning a unique associated identity to each of a plurality of the containers, wherein the identity includes at least one of IP address, host name, and MAC address. For example, Kubernetes containers have an associated hostname, which in the case of a single-container Pod is the unique identity of that container. For another example, Kubernetes pods have an associated hostname, which is unique. For another example, a networked Kubernetes pod has an assigned IPv4 and/or IPv6 address. For another example, a Docker container has an IP address and a hostname. *See, e.g.*:

Claim 6	Accused Instrumentalities
	 Container information The *hostname* of a Container is the name of the Pod in which the Container is running. It is available through the `hostname` command or the `gethostname` function call in libc. The Pod name and namespace are available as environment variables through the downward API. User defined environment variables from the Pod definition are also available to the Container, as are any environment variables specified statically in the container image. https://kubernetes.io/docs/concepts/containers/container-environment/

Claim 6	Accused Instrumentalities
	 IP address and hostname By default, the container gets an IP address for every Docker network it attaches to. A container receives an IP address out of the IP subnet of the network. The Docker daemon performs dynamic subnetting and IP address allocation for containers. Each network also has a default subnet mask and gateway. You can connect a running container to multiple networks, either by passing the `--network` flag multiple times when creating the container, or using the `docker network connect` command for already running containers. In both cases, you can use the `--ip` or `--ip6` flags to specify the container's IP address on that particular network. In the same way, a container's hostname defaults to be the container's ID in Docker. You can override the hostname using `--hostname`. When connecting to an existing network using `docker network connect`, you can use the `--alias` flag to specify an additional network alias for the container on that network. https://docs.docker.com/network/

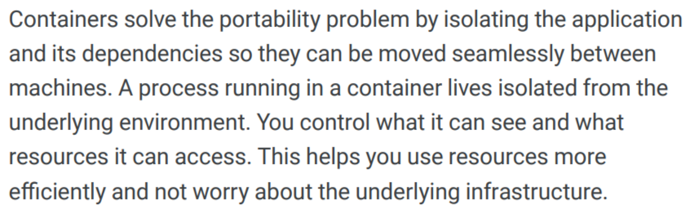
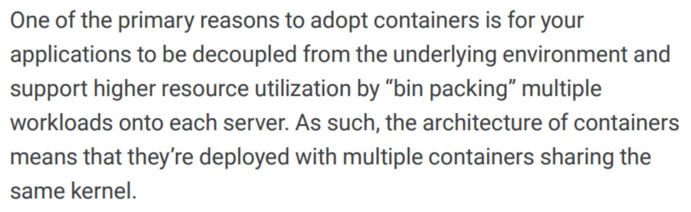
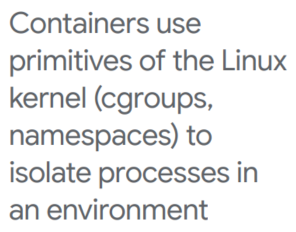
Claim 9

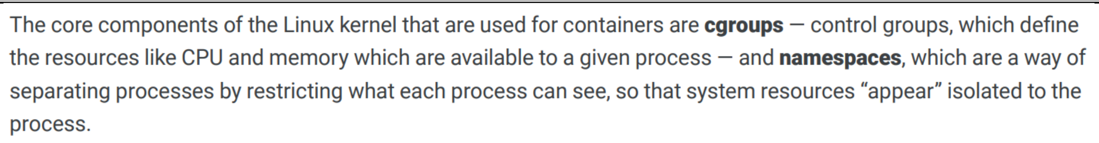
Claim 9	Accused Instrumentalities
9. A method as defined in claim 2, wherein server information related to information related to hardware resource usage including at least one of CPU memory, network bandwidth, and disk allocation is associated with at least some of the containers prior to the	Google and/or its customer practices, through the Accused Instrumentalities, a method as defined in claim 2, wherein server information related to hardware resource usage including at least one of CPU memory, network bandwidth, and disk allocation is associated with at least some of the containers prior to the applications within the containers being executed. For example, Kubernetes tracks and limits resource usage, including CPU and memory resources. For another example, Docker tracks and limits resource usage, including CPU and memory resources. *See, e.g.*:

Claim 9	Accused Instrumentalities
applications within the containers being executed.	 **Resource Management for Pods and Containers** When you specify a Pod, you can optionally specify how much of each resource a container needs. The most common resources to specify are CPU and memory (RAM); there are others. When you specify the resource *request* for containers in a Pod, the kube-scheduler uses this information to decide which node to place the Pod on. When you specify a resource *limit* for a container, the kubelet enforces those limits so that the running container is not allowed to use more of that resource than the limit you set. The kubelet also reserves at least the *request* amount of that system resource specifically for that container to use. Requests and limits If the node where a Pod is running has enough of a resource available, it's possible (and allowed) for a container to use more resource than its `request` for that resource specifies. However, a container is not allowed to use more than its resource `limit`. For example, if you set a `memory` request of 256 MiB for a container, and that container is in a Pod scheduled to a Node with 8GiB of memory and no other Pods, then the container can try to use more RAM. If you set a `memory` limit of 4GiB for that container, the kubelet (and container runtime) enforce the limit. The runtime prevents the container from using more than the configured resource limit. For example: when a process in the container tries to consume more than the allowed amount of memory, the system kernel

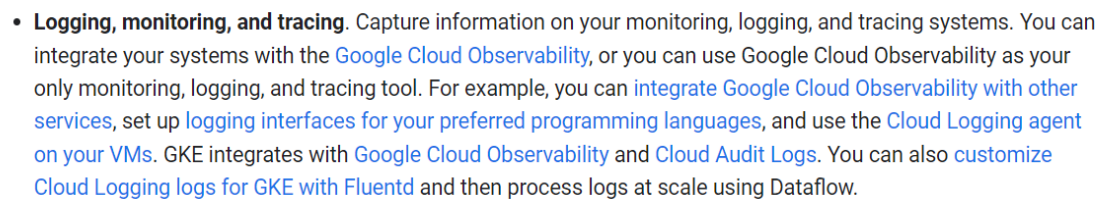
Claim 9	Accused Instrumentalities
	terminates the process that attempted the allocation, with an out of memory (OOM) error. Limits can be implemented either reactively (the system intervenes once it sees a violation) or by enforcement (the system prevents the container from ever exceeding the limit). Different runtimes can have different ways to implement the same restrictions. https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/ **Runtime options with Memory, CPUs, and GPUs** By default, a container has no resource constraints and can use as much of a given resource as the host's kernel scheduler allows. Docker provides ways to control how much memory, or CPU a container can use, setting runtime configuration flags of the `docker run` command. This section provides details on when you should set such limits and the possible implications of setting them. **Limit a container's access to memory** Docker can enforce hard or soft memory limits. • Hard limits lets the container use no more than a fixed amount of memory. • Soft limits lets the container use as much memory as it needs unless certain conditions are met, such as when the kernel detects low memory or contention on the host machine. https://docs.docker.com/config/containers/resource_constraints/

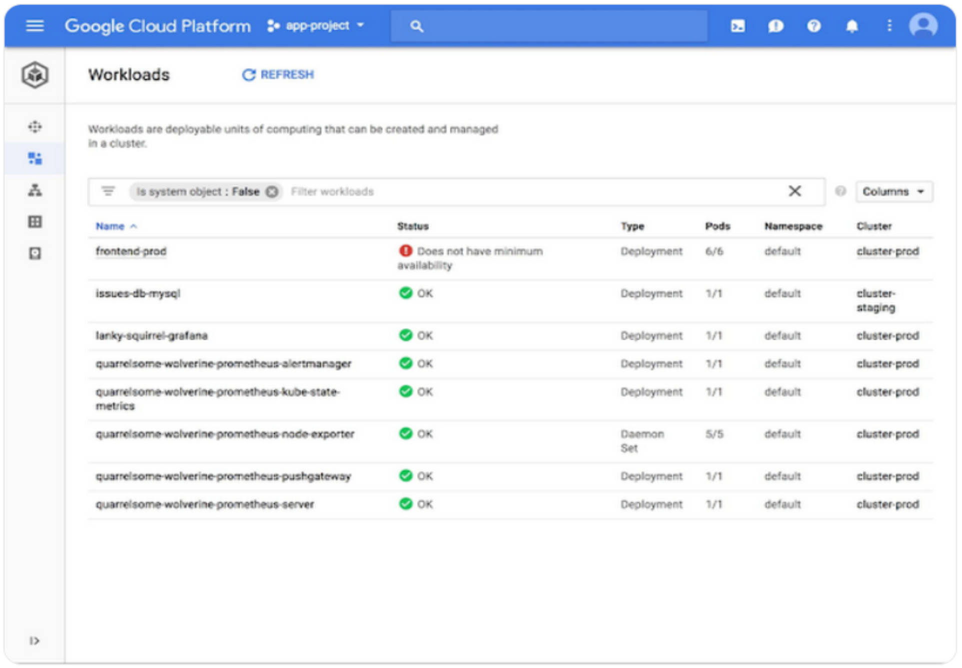
Claim 10

Claim 10	Accused Instrumentalities
10. A method as defined in claim 2, wherein in operation when an application residing within a container is executed, said application has no access to system files or applications in other containers or to system files within the operating system during execution thereof.	Google and/or its customer practices, through the Accused Instrumentalities, a method as defined in claim 2, wherein in operation when an application residing within a container is executed, said application has no access to system files or applications in other containers or to system files within the operating system during execution thereof. As described in connection with limitations [1a] and [1f] above, containers cannot access the host/node's file system, or the file systems of other containers, unless specifically configured to allow such access, for example using a shared volume. *See, e.g.*:  Containers solve the portability problem by isolating the application and its dependencies so they can be moved seamlessly between machines. A process running in a container lives isolated from the underlying environment. You control what it can see and what resources it can access. This helps you use resources more efficiently and not worry about the underlying infrastructure.  One of the primary reasons to adopt containers is for your applications to be decoupled from the underlying environment and support higher resource utilization by "bin packing" multiple workloads onto each server. As such, the architecture of containers means that they're deployed with multiple containers sharing the same kernel.  Containers use primitives of the Linux kernel (cgroups, namespaces) to isolate processes in an environment https://services.google.com/fh/files/misc/why_container_security_matters.pdf

Claim 10	Accused Instrumentalities
	 The core components of the Linux kernel that are used for containers are **cgroups** — control groups, which define the resources like CPU and memory which are available to a given process — and **namespaces**, which are a way of separating processes by restricting what each process can see, so that system resources "appear" isolated to the process. https://services.google.com/fh/files/misc/why_container_security_matters.pdf

Claim 13

Claim 13	Accused Instrumentalities
13. A method as defined in claim 1 further comprising the step of associating with a plurality of containers a stored history of when processes related to applications within the container are executed for at least one of, tracking statistics, resource allocation, and for monitoring the status of the application.	Google and/or its customer practices, through the Accused Instrumentalities, a method as defined in claim 1 further comprising the step of associating with a plurality of containers a stored history of when processes related to applications within the container are executed for at least one of, tracking statistics, resource allocation, and for monitoring the status of the application. *See* analysis and evidence for claim 1 above. *See also, e.g.*:  • **Logging, monitoring, and tracing**. Capture information on your monitoring, logging, and tracing systems. You can integrate your systems with the Google Cloud Observability, or you can use Google Cloud Observability as your only monitoring, logging, and tracing tool. For example, you can integrate Google Cloud Observability with other services, set up logging interfaces for your preferred programming languages, and use the Cloud Logging agent on your VMs. GKE integrates with Google Cloud Observability and Cloud Audit Logs. You can also customize Cloud Logging logs for GKE with Fluentd and then process logs at scale using Dataflow. https://cloud.google.com/architecture/migrating-containers-kubernetes-gke

Claim 13	Accused Instrumentalities
	 https://cloud.google.com/blog/products/gcp/manage-google-kubernetes-engine-from-cloud-console-dashboard-now-generally-available

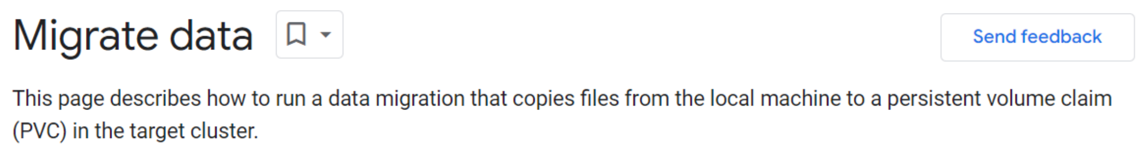
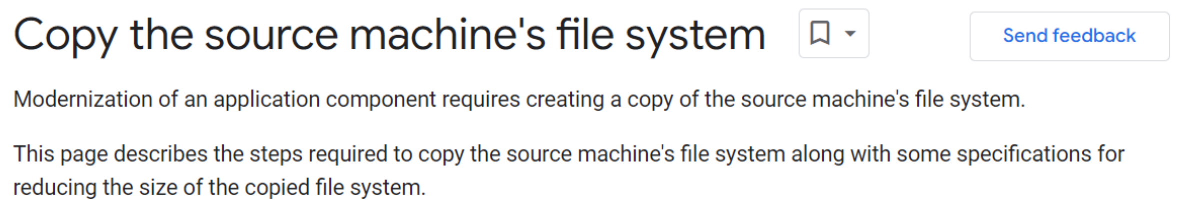
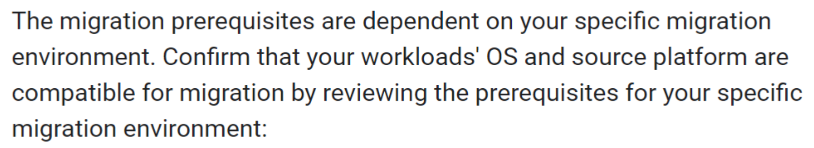
Workloads are deployable units of computing that can be created and managed in a cluster.

Is system object : False — Filter workloads

Name	Status	Type	Pods	Namespace	Cluster
frontend-prod	Does not have minimum availability	Deployment	6/6	default	cluster-prod
issues-db-mysql	✓ OK	Deployment	1/1	default	cluster-staging
lanky-squirrel-grafana	✓ OK	Deployment	1/1	default	cluster-prod
quarrelsome-wolverine-prometheus-alertmanager	✓ OK	Deployment	1/1	default	cluster-prod
quarrelsome-wolverine-prometheus-kube-state-metrics	✓ OK	Deployment	1/1	default	cluster-prod
quarrelsome-wolverine-prometheus-node-exporter	✓ OK	Daemon Set	5/5	default	cluster-prod
quarrelsome-wolverine-prometheus-pushgateway	✓ OK	Deployment	1/1	default	cluster-prod
quarrelsome-wolverine-prometheus-server	✓ OK	Deployment	1/1	default	cluster-prod

<u>Claim 14</u>

Claim 14	Accused Instrumentalities
14. A method as defined in claim 1 comprising the step of creating containers prior to said step of storing containers in memory, wherein containers are created by:	Google and/or its customer practices, through the Accused Instrumentalities, a method as defined in claim 1 comprising the step of creating containers prior to said step of storing containers in memory, wherein containers are created by (a) running an instance of a service on a server; (b) determining which files are being used; and, (c) copying applications and associated system files to memory without overwriting the associated system files so as to provide a second instance of the applications and associated system files.

Claim 14	Accused Instrumentalities
a) running an instance of a service on a server; b) determining which files are being used; and, c) copying applications and associated system files to memory without overwriting the associated system files so as to provide a second instance of the applications and associated system files.	For example, GKE, Cloud Run, and Migrate to Containers support the creation of containers and deploying the containers on the server. The containers are first created (e.g., by Migrate to Containers) and then later deployed/stored on the server (e.g., by GKE or Cloud Run). The creation step involves determining which applications and files are to be migrated, copying these identified applications and files to a location in the target server. Based on information and belief, once the files are migrated, the earlier stored files (if any) are not deleted/overwritten, rather, the migrated files are stored as different instance in memory accessible to containers. Further, an instance of an application/service may be tested or run on the target server to ensure compatibility. *See* analysis and evidence for claim 1 above. *See also, e.g.*:  Migrate data This page describes how to run a data migration that copies files from the local machine to a persistent volume claim (PVC) in the target cluster. https://cloud.google.com/migrate/containers/docs/m2c-cli/migrate-data  Copy the source machine's file system Modernization of an application component requires creating a copy of the source machine's file system. This page describes the steps required to copy the source machine's file system along with some specifications for reducing the size of the copied file system. https://cloud.google.com/migrate/containers/docs/m2c-cli/copy-file-system  The migration prerequisites are dependent on your specific migration environment. Confirm that your workloads' OS and source platform are compatible for migration by reviewing the prerequisites for your specific migration environment: https://cloud.google.com/migrate/containers/docs/setting-up-overview